The Copland Toolbox

Preliminary

Developer Press

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Introduction to the Copland Toolbox

Contents

The Copland Toolbox consists of system software services that you can use to create your application's human interface elements and present them to users. The Toolbox also simplifies a variety of human interface programming tasks and provides low-level support for active assistance.

This chapter introduces the Copland Toolbox. "Creating an Application's Human Interface," beginning on page 1-6, introduces the Copland event-handling model and the standard human interface elements provided by the Toolbox. "Copland Toolbox Architecture," beginning on page 1-40, introduces key programming concepts that underlie the Toolbox.

Chapter 2, "The Toolbox: System 7 Compared With Copland," provides information for System 7 developers who want to begin planning for Copland.

▲ **WARNING**
This document is preliminary and incomplete. All information presented here is subject to change in later developer releases. ▲

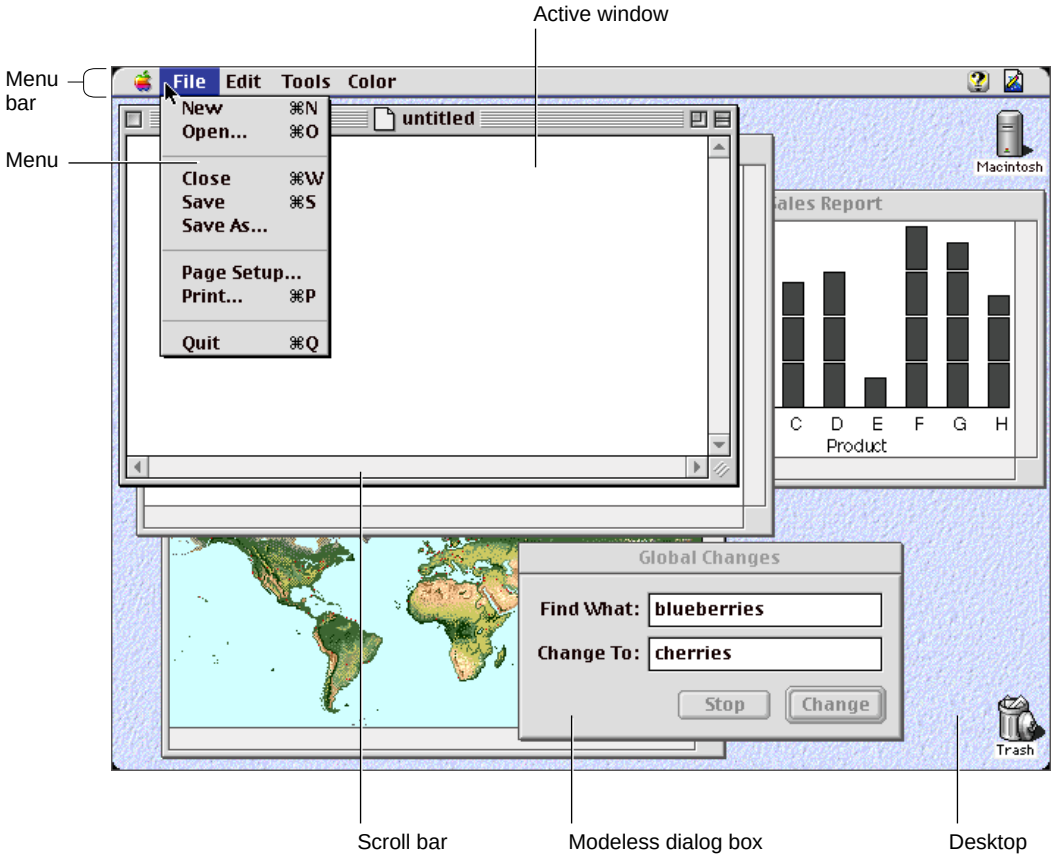
A typical Copland application presents users with a carefully designed visual interface that allows them to perform actions and accomplish goals according to their own priorities. To ensure that human interface elements share consistent behavior and appearance across all applications, the Copland Toolbox provides a comprehensive set of standard interface elements that you can piece together according to your application's needs. This ensures that common elements such as pop-up menus and sliders work the same way in different applications and coordinates with the appearance of other elements.

The Copland Toolbox also supports customization by each user in ways that maintain the overall look and feel of the human interface for that user. For example, users can choose among different themes, or styles—that is, coordinated sets of human interface designs that determine the appearance of human interface elements on a systemwide basis, across multiple applications.

The figures that follow show some of the standard human interface elements provided by the Toolbox and the way their appearance changes when the user switches themes. Figure 1-1 shows how the screen might appear when a user has selected the Apple Default theme and is interacting with a typical Mac OS–compatible application, called SurfWriter, that permits simple text editing.

Note
Unless otherwise indicated, the human interface elements illustrated in this chapter use the Apple Default theme. ◆

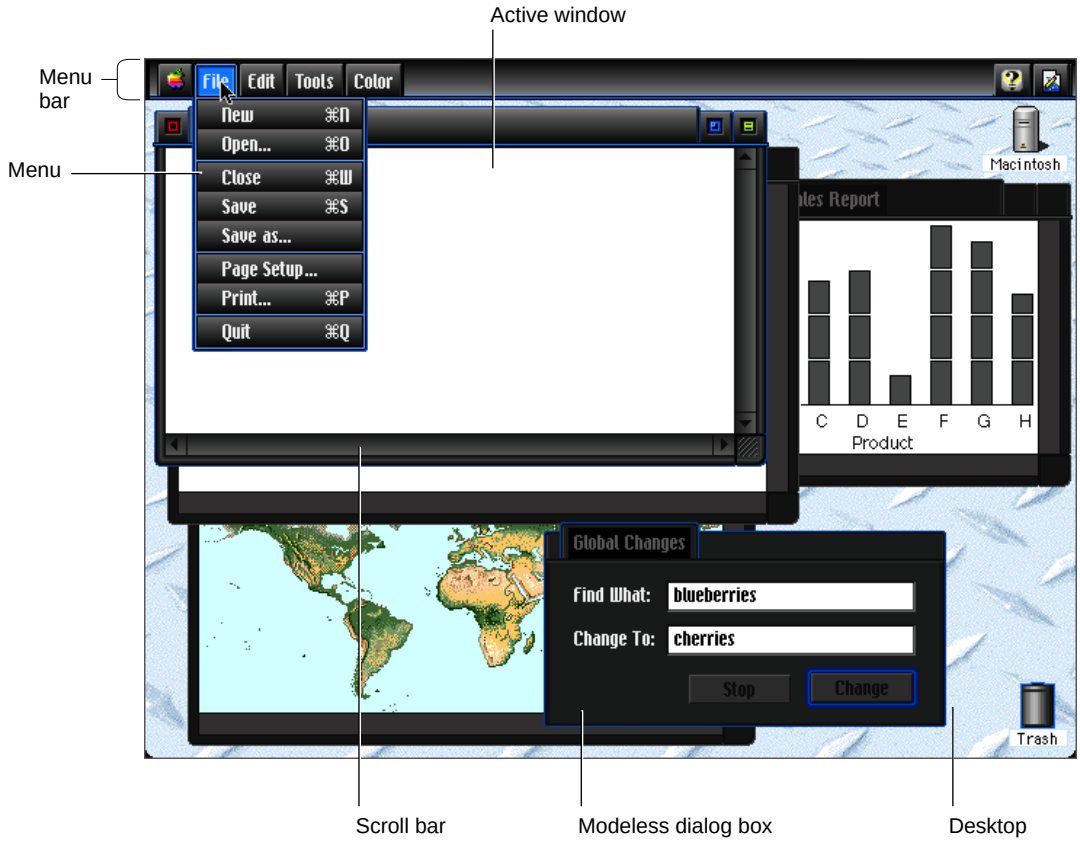
Figure 1-1 The SurfWriter application as it appears in the Apple Default theme



A user can directly control the SurfWriter application by means of a variety of human interface elements, including

- menus that allow the user to choose commands
- windows that allow the user to enter and edit information
- scroll bars and other controls that the user can manipulate
- dialog boxes that solicit information from the user

Figure 1-2 The SurfWriter application as it appears in an alternate theme



In addition to interacting freely with the application by manipulating its visual interface, the user can change the appearance of all windows, controls, menus, and other elements displayed on a single computer by choosing a different theme. Figure 1-2 shows the SurfWriter application as it appears in an alternate theme.

Apple supplies several standard themes. The Apple Default theme shown in Figure 1-1 is built into the system. Users can install additional themes, switch installed themes, or remove installed themes whenever they wish.

Creating an Application's Human Interface

The Copland Toolbox provides a complete programming model for creating an application's human interface. From the user's point of view, a Copland application

- responds to user actions, such as mouse actions or keyboard input
- displays windows, alert boxes, and dialog boxes that present data and various choices about manipulating the data to the user
- displays controls that let users manipulate the application directly with a pointing device such as a mouse
- displays menus that let the user choose from lists of choices or commands

In general, the user should always be free to choose the next action to perform. To support this freedom in a Copland application, you use the Apple Event Manager, which provides a systemwide mechanism for distributing events in a preemptively safe manner. Events that use this mechanism are called **Apple events.** By default, the Toolbox interprets Apple events as they are delivered by the Apple Event Manager and directs them to the appropriate menu or window.

To create your application's windows, you use the Window Manager. To create most of the other elements of your application's human interface, including its dialog boxes, controls and menus, you use the Panels class library.

A **panel** is a SOM object that encapsulates one or more human interface elements. The Panels class library defines a wide range of standard elements commonly used in applications, including menus, dialog boxes and alert boxes, scroll bars and other controls, lists, icons, and visual separators. Although panels can be used within OpenDoc parts, they aren't intended to be as large or as powerful as parts. Instead, they facilitate the assembly of integrated human interface elements from smaller, simpler objects.

One especially useful kind of panel is an **embedding panel,** which contains other panels (possibly including other embedding panels) and distributes events to them. For example, everything inside a standard Copland window is usually contained in an embedding panel. You use embedding panels and the other standard panels defined by the Panels class library to assemble your application's visual interface. If necessary, you can also define your own custom panels using object-oriented programming (OOP) techniques.

In addition to providing windows, dialog boxes, menus, and other basic human interface elements, a Copland application

- has characteristic icons that represents the application file and the application's documents in the Finder
- lets users specify application-specific preferences
- supports copy, paste, and drag and drop

To implement these capabilities in your application, you use the Finder interface, the Preferences Manager, and the Scrap, Clipboard, and Drag Managers.

You generally specify human interface elements for your application in resources that are completely independent of your application's source code. This greatly simplifies localization for different countries and languages. The Resource Manager provides functions for managing resources.

The sections that follow introduce some of the standard features of Copland applications and the Toolbox services you use to implement them. For more information about the architectural principles shared by all Toolbox services, see "Copland Toolbox Architecture," beginning on page 1-40.

Events

System 7 and earlier versions of the Mac OS require applications to have an event loop, a piece of code that continually polls the system for events and responds to those events appropriately. Although this arrangement allows the user considerable freedom in choosing when to perform various actions, it has limitations in the preemptive multitasking environment provided by Copland. Copland introduces a new event model, based on the Apple events mechanism introduced in System 7, that provides a unified interface for events throughout the system, avoids the problems created by polling, and enhances application responsiveness to user actions.

From a user's point of view, the way Copland handles events is similar to the way the Mac OS has always handled them. For example, the user can type text in a window, select a graphic and copy it, open a new document in a different application, paste in the graphic, open another document, then go back to the first window to select text and change its size, style, or font.

From a programmer's point of view, the Copland event model differs significantly from the event loop model. The essence of Copland event

handling is simple. When the user launches your application, it informs the Apple Event Manager what kinds of events it's interested in receiving. Your application then informs the Apple Event Manager that it is ready to receive events, and the Apple Event Manager blocks the calling task until an event in which your application has expressed an interest arrives. This arrangement takes maximum advantage of priority-based preemptive scheduling, allowing other applications and tasks to receive processing time when your application doesn't need it.

To identify which events it's interested in receiving, your application installs event handlers in one or more **event handler tables,** which the Apple Event Manager uses to dispatch events as the application receives them. In addition, the Toolbox provides a set of default event handlers that automatically interpret low-level events such as mouse clicks and keypresses and pass them on to the appropriate window or menu item. Your application can override any of the default event handlers.

When a window receives an event, it processes the event on the basis of the window's own **window event handler.** If the event is in the title bar or some other area controlled by the Window Manager, the default window event handler responds to the event; otherwise, it sends the event on to the **root panel,** which is an embedding panel that contains all of the window's other panels. As with other handlers, your application can override the default window event handler if necessary.

When the root panel inside a menu or a window receives an event, it handles the event by calling methods of the subpanels it contains. "Panels," beginning on page 1-13, describes the Panels class library and how you can use it to implement your application's human interface.

The best way to support the Copland event model is to separate the code that controls your application's user interface from the code that responds to the user's manipulation of the interface. This is called **factoring** your application. A fully factored application translates user actions into Apple events that the application sends to itself to handle those actions appropriately. Factoring not only supports the Copland event model, but also allows your application to be controlled by means of any scripting language, such as AppleScript, that's based on the Open Scripting Architecture (OSA).

Another difference between the Copland event model and a traditional event loop involves **periodic processing,** which is processing that takes place at specified intervals. For example, if the user isn't doing anything else, an application should be able to perform repetitive tasks such as making the caret

blink in the active window. To support this kind of processing in Copland, you use periodic Apple events.

Periodic processing is different from **background processing,** which takes place in the background while the user continues to work. You perform background processing by creating a secondary task that executes preemptively and concurrently while the primary task that controls the human interface continues to respond to user actions.

For example, it may be desirable for a graphics application to perform intensive calculations related to image processing in the background, allowing the user to continue to interact with the application's human interface without loss of responsiveness. When the calculations are complete, the secondary task can transfer the result of the calculations to the primary task, which actually draws the image to the screen.

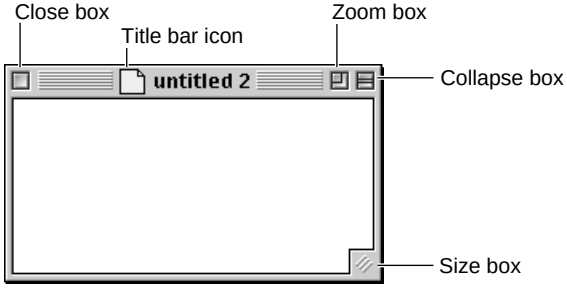
For more information about the Copland tasking model, see the document "Kernel and Operating System Services."

For more information about the Copland event model, see "The Copland Event Model," beginning on page 2-5 of this document.

Windows

Most applications use windows to present information to and interact with the user. Figure 1-3 shows a standard document window and its elements.

Figure 1-3 A standard document window



The window in Figure 1-3 includes the following elements:

- A **close box** that dismisses the window.
- A **title bar icon** that users can drag to a new volume to copy the document to that volume. The behavior of the title bar icon is determined by you within guidelines to be provided by Apple.
- A **collapse box** in the upper-right corner that users can click to control "windowshade" behavior—that is, to hide or show all of the window except the title bar.
- A **zoom box** next to the collapse box. The Copland Window Manager includes built-in support for monitor-specific zooming; that is, clicking the zoom box causes the window to expand so it fills the screen of the monitor on which it is displayed.
- A **size box** in the lower-right corner that users can drag to resize the window.

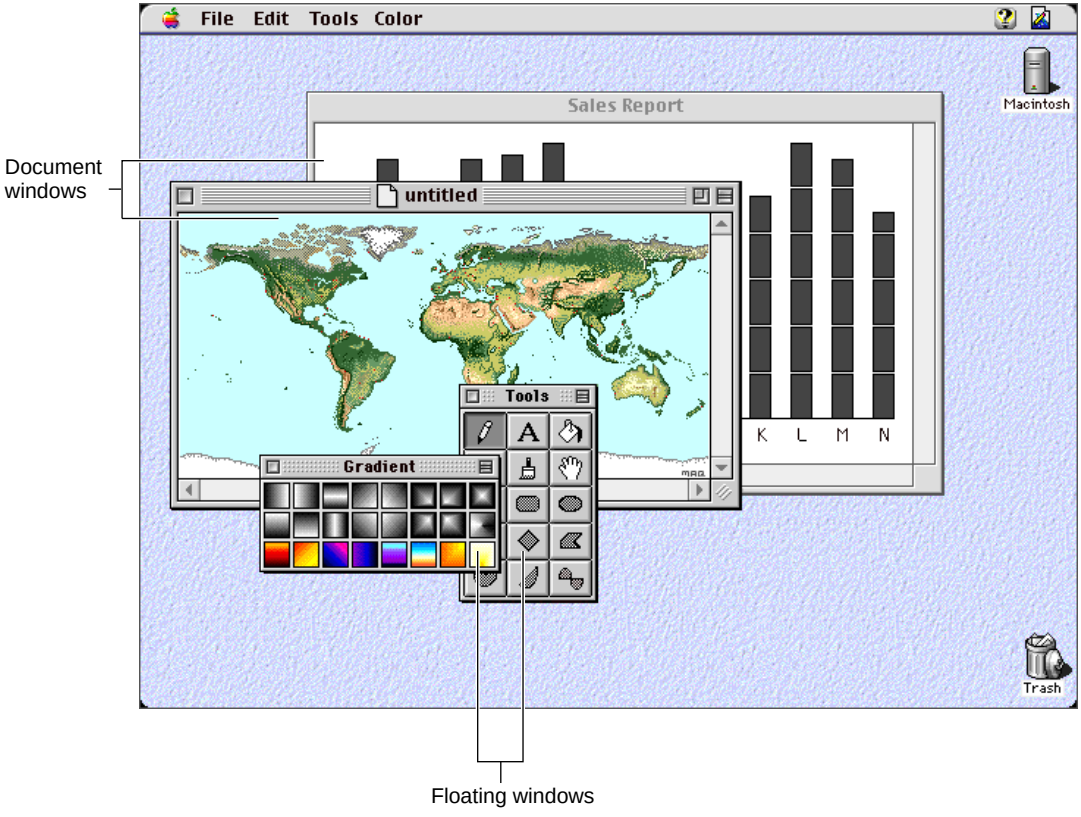
The Window Manager allows you to specify various attributes of any window, such as whether it has a size box, collapse box, zoom box, title bar icon, or close box. It also supports resizing of windows in directions other than down and to the right and the use of text objects in window titles, both of which greatly simplify localization. (For more information about Toolbox support for text objects, see "International Text" on page 1-43.)

Every window belongs to one of three classes:

- **Modal windows** appear in front of all other kinds of windows in an application's layer. They are used for dialog boxes and alert boxes that require immediate attention from the user.
- **Floating windows** appear in front of document windows and behind modal windows in an application's layer. They are used for tool palettes, catalogs, and other elements used to act on data in document windows.
- **Document windows** (like that shown in Figure 1-3) appear behind floating windows and modal windows in an application's layer. They are used for document data such as graphics and text.

The Window Manager keeps windows of each class in separate sublayers within a single application's layer. For example, the floating windows shown in Figure 1-4 always appear in front of the same application's document windows.

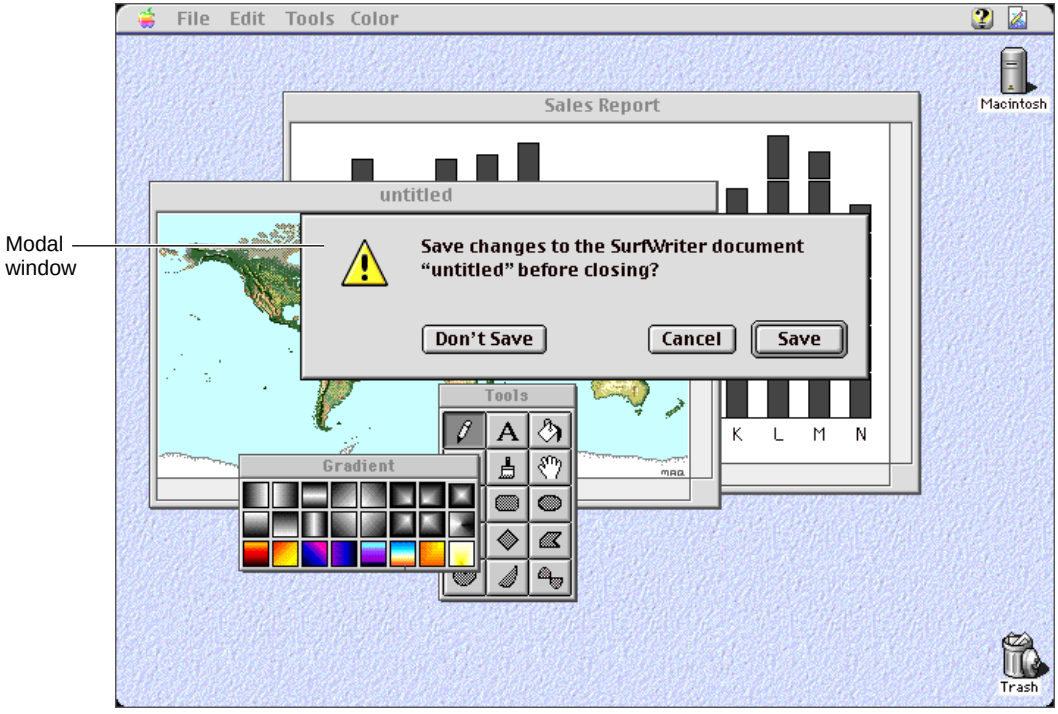
Figure 1-4 Layering of floating windows and document windows



If the user performs some action that invokes a modal window, such as attempting to close a document without saving it, the modal window appears in front of all other windows in the application's layer, as shown in Figure 1-5.

When your application displays any modal window, the menu bar automatically changes to a modal state. The menu bar returns to its original state when the modal window is gone.

Figure 1-5 The Window Manager ensures that modal windows always appear in front of other windows in an application's layer.



As shown in the preceding figures, a user typically has one or more windows open on the desktop, often from several different applications. However, only one window can be the active window. The **active window** is the window that appears frontmost on the desktop, and it is identified by distinctive details that aren't visible for inactive windows. For example, in Figure 1-4, the document window "untitled" is active, and the other document window is inactive; whereas in Figure 1-5, the modal window is the active window, and both document windows are inactive.

All keyboard activity is directed toward the active window. Make sure your application follows the human interface guidelines regarding active and inactive windows. For example, you should show the scroll bars and highlight any selection in an active window, and hide the scroll bars and change or remove highlighting from any selection in an inactive window.

The chapter "Window Manager," which will be available with later developer releases, describes the standard kinds of windows and how to create, move, size, zoom, or update the contents of your window using Window Manager functions.

Panels

The Panels class library provides a uniform, object-oriented interface for implementing human interface elements commonly used by applications. These standard elements are based on SOM classes that all ultimately inherit from the abstract superclass Panel, as shown in Figure 1-6.

▲ **WARNING**
The class names and relationships shown in Figure 1-6 are preliminary and subject to change in later releases. ▲

Figure 1-6 Top two levels of the inheritance hierarchy for the Panels class library

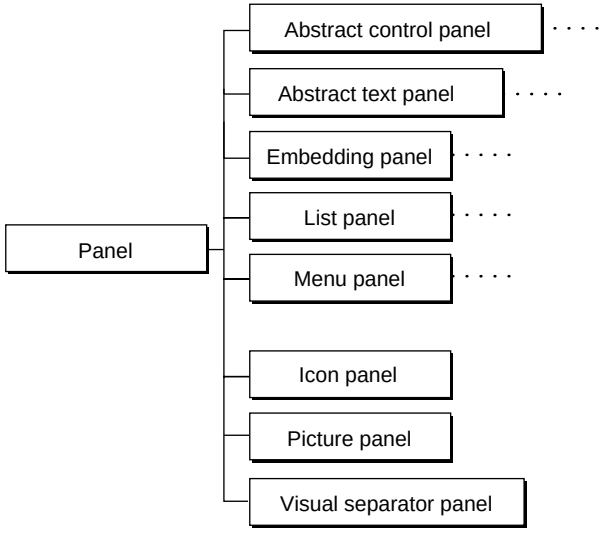


Figure 1-6 shows preliminary plans for the top two levels of the Panels inheritance hierarchy (which also includes additional subclasses not shown in the figure). These classes play the following roles:

- The panel class is the superclass for all panel classes, and as such it defines the attributes and methods shared by all panels. These include methods for initializing, enabling or disabling, drawing, and hiding or showing a panel, as well as methods that control keyboard focus, mouse interaction, copy, paste, drag and drop, and other standard interactive behavior. Other classes in the Panels class library define additional methods and other characteristics as necessary.
- The abstract control panel class is the superclass for all controls, including sliders, scroll bars, pop-up menus, progress indicators, and buttons such as checkboxes and radio buttons.
- The abstract text panel class is the superclass for all textual human interface elements, including static text, styled static text, and editable text.
- The embedding panel class is the superclass for panels that contain other panels, including dialog boxes, alert boxes, and radio button groups.
- The list panel class is the superclass for all panels that encapsulate lists, including icon lists, text lists, and scrolling lists.
- The menu panel class is the superclass for all panels that encapsulate menus.
- The last three classes shown in Figure 1-6 encapsulate simple visual elements that are commonly used in applications. These classes provide a convenient way to integrate purely visual elements with other human interface elements:
  - The icon panel class defines panels that encapsulate icons of different sizes and bit depths.
  - The picture panel class defines panels that encapsulate bitmapped images.
  - The visual separator class defines panels that encapsulate horizontal, vertical, and rectangular visual separators.

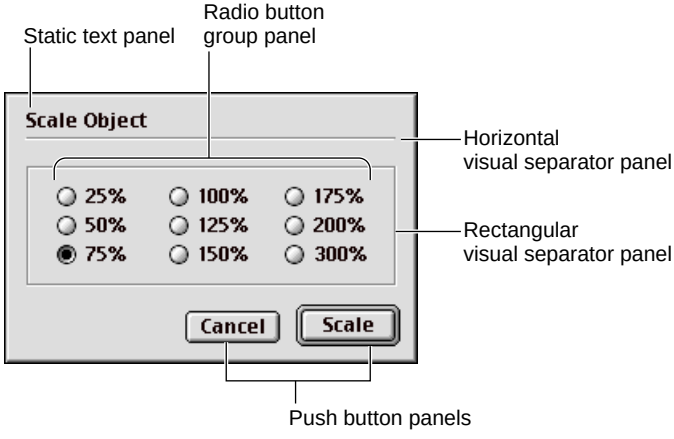
None of the standard panels require a traditional Toolbox manager. Instead of calling procedural functions, you create and manipulate panel objects using SOM techniques. Each panel knows how to draw itself appropriately depending on its state; for example, checkboxes can check and uncheck themselves, sliders can change their appearance in response to dragging, menu panels can highlight correctly when selected, and so on.

A panel controls all aspects of its appearance and behavior as its state changes in response to user activities. Your application determines what effect changes in the panel's state have within the application.

When you instantiate any panel, you can identify a function the panel will call when a state change should generate a specific action by your application. For example, the change in state that occurs when a user chooses a menu command might invoke the function provided by your application that executes the command. This arrangement keeps the implementation of the panel itself separate from the specific behavior that your application associates with a particular panel state.

Figure 1-7 illustrates a typical use of standard panels in a modal dialog box.

Figure 1-7 Standard panels used in a modal dialog box



The dialog box shown in Figure 1-7 includes these standard panels:

- **Dialog box panel.** All the panels in Figure 1-7 are embedded within the standard dialog box panel, which is a special kind of embedding panel that encapsulates the entire contents of a dialog box. The dialog box panel isn't labeled in the figure; it consists of the entire content area of the modal dialog box. Like any other embedding panel, a dialog box panel controls the layout, keyboard focus, and mouse interaction for all the panels it contains.
- **Static text panel.** A panel that displays static text.

- **Radio button group panel.** A specialized embedding panel that encapsulates radio button controls. A radio button group panel controls keyboard focus and mouse interaction for its radio buttons, changing their highlighting and state as appropriate in response to mouse clicks and keypresses.
- **Horizontal and rectangular visual separator panels.** Visual separator panels can be manipulated and positioned like any other panel within any embedding panel. You can also specify title text for the rectangular visual separator to display, as in the movable modal dialog box in Figure 1-11 on page 1-23.
- **Push button panels.** Keyboard focus for the Cancel and Scale button panels is integrated with the rest of the panels, and they highlight themselves correctly when clicked.

You can place panels in any window, not just a dialog box, and you can combine panels to create toolbars and other complex human interface elements, including menus.

Like any SOM-based class library, the Panels class library provides three key benefits associated with OOP:

- **Inheritance** allows subclasses to make use of characteristics defined by classes above them in their branches of the class hierarchy. You can use SOM subclassing techniques to derive new panel classes from standard panel classes, creating new kinds of elements without having to start from scratch.
- **Encapsulation** refers to the packaging of all the code that implements a panel's human interface behavior within the panel object itself, thus protecting it from accidental or inappropriate changes. Panels are autonomous SOM objects that you can reuse in completely unrelated applications. You don't have to subclass each time you want to use a particular kind of panel for a new purpose. Instead, you simply specify a different function during instantiation that implements application-specific behavior when the panel's state changes.
- **Unification** of disparate human interface elements in a single programming interface simplifies the construction of your application's human interface. You always implement common behavior, such as showing or hiding a panel, the same way no matter what kind of panel is involved. Instead of learning how to implement similar behaviors in slightly different ways for different managers, you use the same method to implement the same behavior for any panel.

For more information about SOM, see the document "SOM and Software Extensibility in Copland."

You don't need to use an object-oriented language such as C++ to take advantage of these benefits. Some familiarity with OOP is required only if you subclass from the standard classes; and you need to subclass only if you want to create panels whose appearance or behavior differ from the standard panels. For a discussion of custom panels, see "Customizing Panels," beginning on page 1-44.

In addition to the benefits of OOP, the Panels class library includes support for the following capabilities:

- **Binary compatibility.** As SOM objects, future versions of any panel can be released without breaking existing versions.
- **Embedding.** You can arrange standard panels in containment hierarchies by using various kinds of embedding panels, without any need for subclassing.
- **Keyboard navigation.** Panels know how to react to keystrokes and how to display themselves with and without keyboard focus.
- **Collection items.** You can attach collection items to any panel. (For more information about collections, see "Extensible Data Structures" on page 1-43.)
- **Drag Manager support.** You can support drag and drop for any panel just by overriding a few methods.

The sections that follow include examples of some of the standard panels defined by the Panels class library.

Controls

Most windows and dialog boxes contain controls. Controls are human interface elements that the user can manipulate with the mouse to perform actions in your application or to change settings that modify future actions.

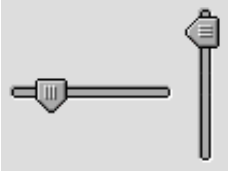
Table 1-1 shows examples of the panels you can create with the panel classes for controls. The appearance of each control is defined by its class. The current theme determines details of its appearance, and your application determines its actions and settings.

You can use panel methods to draw most of the standard controls with **keyboard focus**—that is, change a control's appearance as appropriate when it is the focal point on screen for actions triggered by keypresses.

Table 1-1 Some of the standard panels that encapsulate controls

Example in the Apple Default theme	Name	Description
	Push button panel	A button that displays text indicating its purpose. Used to perform an instantaneous action when clicked by the user, such as completing operations defined by a dialog box or acknowledging an error message in an alert box. You can optionally specify that a push button is the default button, in which case it draws itself with standard default appearance for the current theme; for example, with a ring around it.
	Icon button panel	A button that displays an icon. Used in situations where icons are more convenient than text for indicating a button's purpose, for example in palettes and toolbars. An icon button can animate momentarily like a push button or toggle back and forth between a pressed state and an unpressed state, similar to a checkbox. Icon buttons come in three sizes: small, medium, and large. This example shows a large icon.
	Checkbox panel	A button that displays a small square with accompanying text indicating what kind of setting the checkbox controls. Used for an option that must be off, on, or in a mixed state. In the Apple Default theme, the square contains an X when the setting associated with the box is on, is empty when the setting is off, or contains a short horizontal line when the setting is mixed. (For more information about the mixed state, see page 1-21.)

Example in the Apple Default theme	Name	Description
	Radio button panel	A button that displays a circle with a title beside it indicating what kind of setting the radio button controls. Like checkboxes, radio buttons retain and display an on-or-off setting; however, only one radio button in a group of radio buttons should be on at any one time. In the Apple Default theme, the circle is filled when the setting associated with the button is on, is empty when the setting is off, or contains a short horizontal line when the setting is mixed. (For more information about the mixed state, see page 1-21.)
	Disclosure triangle panel	A triangle used to control progressive disclosure in lists, such as lists of files and folders in the Finder. When the arrow points right, only one item should be visible beside it. When the arrow points down, both the original item and the items contained within it should be visible in the list. To toggle between the two states, the user clicks the disclosure triangle, which turns with a characteristic animation defined by the current theme.
	Little arrows panel	A pair of arrows that typically accompany a text box containing a numerical value, such as the date or time. Clicking the up arrow should increase the value in the text box, and clicking the down arrow should decrease it. The new values don't need to be contiguous to the old ones as long as they change in a logical and predictable manner appropriate for whatever they represent.

Example in the Apple Default theme	Name	Description
	Progress indicator panel	A horizontal display used to indicate the progress of a lengthy operation (typically more than three seconds). If you don't know how long an operation will take, you can let the user know that it's still in progress by rotating a progress indicator like a barber pole, as shown in the upper example. If you can supply values to the panel indicating how much of an operation has been completed, the progress indicator can fill itself in from one end to the other to indicate what percentage of the operation has been completed, as shown in the lower example.
	Slider panel	Displays a range of values, magnitude, or position. A movable indicator shows the current setting. Some sliders allow users to alter the value of the slider by moving the indicator up and down or back and forth. Sliders can be analog or digital devices that display their values graphically.
	Pop-up menu panel	A menu that appears in a dialog box or window. Used as an alternative to a radio button group or a list, a pop-up menu allows the user to select from several choices. When the user presses the mouse button while the pointer's over the menu title, additional menu items appear, as shown in this example. Pop-up menu panels can support all the features of regular menus, including icons and other menu item types, sticky menus, and tear-off menus. They also handle mouse and keyboard interaction, including highlighting the menu item and tracking keyboard focus.
	Scroll bar panel	A narrow rectangle with an arrow in a box at each end and a scroll box that moves between them. Windows can have a horizontal scroll bar, a vertical scroll bar, or both. Users can click the arrows or drag the scroll box to display more of the document by scrolling it into view. Scrolling should be live—that is, the contents of the window should scroll at the same time that the user is dragging the scroll box.

Radio buttons and checkboxes can be displayed in three different states: on, off, or mixed. A **mixed state** indicates that a setting is on for some elements and off for others. For example, a checkbox that determines whether a character is boldface appears in a mixed state if some characters in a range of selected text are bold and others aren't. Because a mixed state provides feedback about the settings for several elements, the user can't change it directly. Instead, the user must change the settings for the individual elements.

You can use panel methods to get a series of control values back from a control such as a slider or scroll bar while a user is still manipulating it. For example, you can get control values back from a scroll bar that allow your application to redraw the window's contents while the user is dragging the scroll box (live scrolling), or you can change the sound volume while the user is still manipulating the slider rather than waiting until the user releases it. All control values are 32-bit values, permitting manipulation of any control at a very detailed level of granularity.

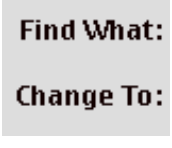
The chapter "Controls," which will be available with later developer releases, describes how to create and manipulate the standard controls.

Text Elements

The Panels class library includes separate classes for static text panels, styled static text panels, and editable text panels.

A static text panel or static styled text panel encapsulates a text object. Figure Figure 1-8 shows two static text panels.

Figure 1-8 Static text panels



An editable text panel encapsulates a text object and permits editing of the text it contains. Editable text panels support keyboard navigation, copy, paste, drag and drop, and styled text. Figure 1-9 shows an editable text panel.

Figure 1-9 An editable text panel

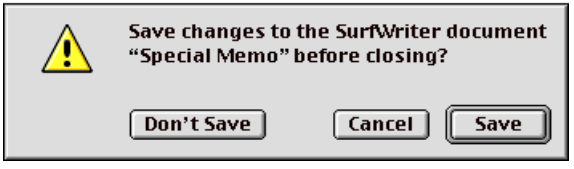
For more information about text objects, see "International Text" on page 1-43.

Alert Boxes and Dialog Boxes

In addition to windows and their contents, a Copland application also uses alert boxes and dialog boxes to communicate with the user. An application displays an **alert box** to warn or to report an error to the user. An alert box typically consists of an icon, text describing the situation, and buttons for the user to acknowledge or rectify the problem.

Figure 1-10 shows an alert box that the SurfWriter application displays when the user attempts to close a window without saving the document. The alert box gives the user a chance to save the document before the SurfWriter application closes the window, thus helping to avoid accidental data loss.

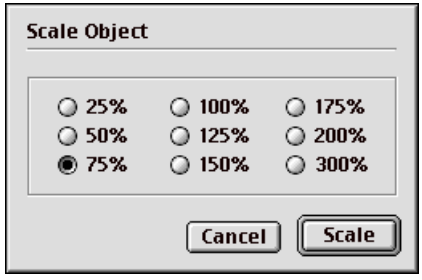
Figure 1-10 An alert box panel



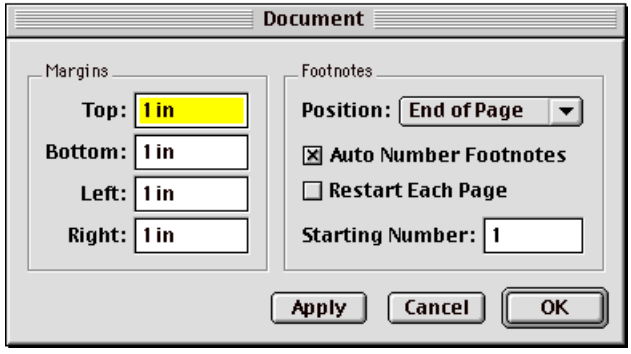
An application displays a **dialog box** to solicit specific kinds of information from the user. Embedding panel classes include methods that you can use to assemble standard panels such as buttons, text panels, lists, and visual separators in alert boxes and dialog boxes. You also use panel methods to handle user interactions with dialog boxes.

You can create modal, movable modal, or modeless dialog boxes. Figure 1-11 shows an example of each type. Each dialog box in the figure is constructed from a variety of standard panels.

Figure 1-11 Modal, movable modal, and modeless dialog box panels



A modal dialog box



A movable modal dialog box

A modeless dialog box

A **modal dialog box** requires the user to work in a single mode—that is, only inside the dialog box—until the completion of the user's interaction with that dialog box. A modal dialog box is similar in appearance to an alert box, except that a modal dialog box can contain any combination of panels, whereas an alert box consists only of text, icon, and button panels. The user cannot move a modal dialog box, and the user can dismiss it only by clicking the appropriate buttons. You should use a modal dialog box only when it's essential for the user to complete an operation before performing any other work.

A **movable modal dialog box** is a modal dialog box with a title bar (but no close box) that allows the user to move the dialog box. The user can dismiss the dialog box only by clicking its buttons; however, when you use movable modal dialog boxes, you should allow the user to switch layers by clicking in another application's window or by choosing another application from the Apple or Application menu. Use a movable modal dialog box when the user might need to move the dialog box to view other areas of the screen or when the user can switch to another application without affecting the state of your application.

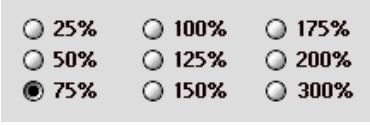
A **modeless dialog box** is a dialog box that looks like a document window without a size box or scroll bars. A modeless dialog box does not require the user to respond before doing anything else. The user can move a modeless dialog box, move between a modeless dialog box and other windows, and close a modeless dialog box just like a document window. Whenever possible, use a modeless dialog box instead of a movable modal or modal dialog box. Use a modeless dialog box when the user can perform other operations—such as working in document windows—without dismissing the dialog box.

The chapter "Dialog Boxes and Alert Boxes," which will be available with later releases, describes how to create alert boxes and dialog boxes.

Radio Button Groups

A radio button group is a special embedding panel that encapsulates several radio button panels, as shown in Figure 1-12. Unlike the individual radio button panel illustrated on page 1-19, a radio button group panel can handle mouse and keyboard interaction, including highlighting and tracking keyboard focus.

Figure 1-12 A radio button group panel



Lists

A **list** is a series of items displayed within a rectangle. Each item in a list is contained within a rectangular **cell.** All cells within a list are the same size, but may contain different types of data and multiple columns of data. The user can click cells to select them. Figure 1-13 shows two simple list panels.

Figure 1-13 Simple list panels

You can create lists with or without scroll bars. To arrange one or more lists with buttons and other controls in a window, you simply add the lists and other panels to an embedding panel.

You can use list panel methods to store and update the data within a list, display the list within a window with an appearance that matches the current theme, and respond appropriately to mouse clicks within a list. List panels store all offsets and values using 32-bit values, permitting the association of large amounts of data with a single list.

The chapter “Lists,” which will be available with later developer releases, describes in detail how to create lists.

Menus

A menu lets the user view or choose an item from a list of choices or commands that your application provides. You design your application's menus according to the tasks or actions your application performs. All applications should support the Apple, File, Edit, Help, Keyboard, and Application menus.

A menu consists of several **menu items.** The menu panel class defines a variety of standard menu items, including text, submenus, icons, and pattern and color cells. Menu panels also allow you to

- create grid menus that contain colors, patterns, and icons
- make any application menu or submenu a tear-off menu
- provide custom content on an item-by-item basis
- display any font in any language using any script in a menu
- display a keyboard equivalent for any menu item using multiple modifier keys
- show and hide the menu bar
- support a "sticky menu" mode that allows users to leave a menu or submenu open and choose menu items by clicking them or from the keyboard

Figure 1-14 and Figure 1-15 show three menus created from standard menu panel classes, including pattern and color swatches, text items based on text objects, dividers, and submenus. You can freely mix icons, text, marks, and keyboard equivalents in any menu item.

The chapter "Menus," which will be available with later developer releases, describes in detail how to create menu panels.

Figure 1-14 Menu created from the standard menu panel classes

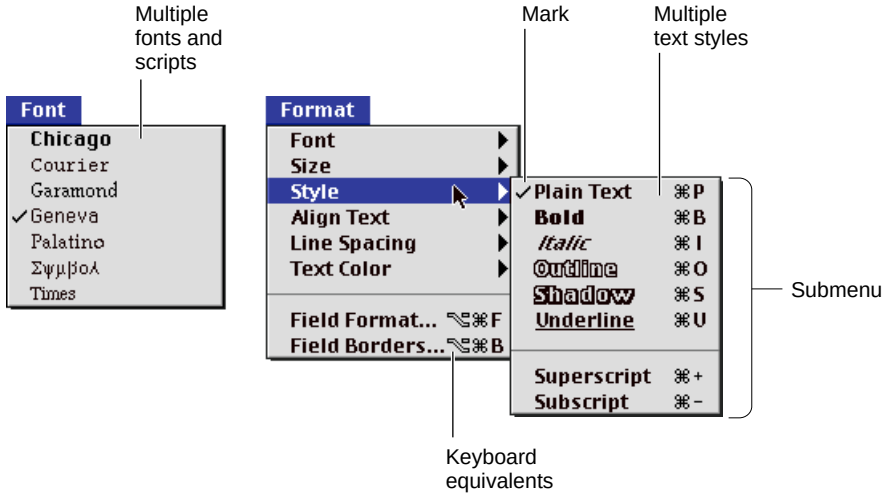
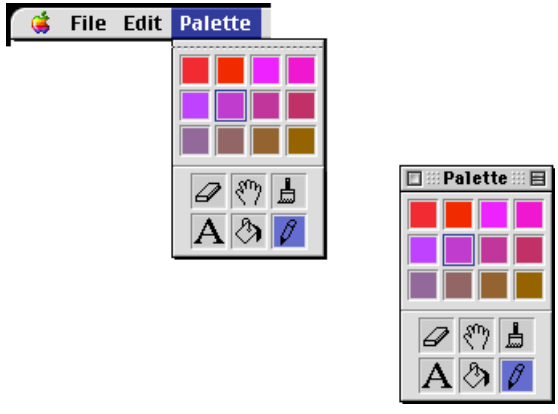


Figure 1-15 Tear-off menu with custom layout



Simple Visual Elements

In addition to the interactive panels introduced in the preceding sections, the Panels class library defines a variety of simple visual panels, including icons, pictures, and visual separators. Although users don't interact with these elements, it is often convenient to implement them as panels.

Simple visual elements can be implemented in two ways: by using the Panels class library panels or by using lower-level services. The visual panels defined by the Panels class library encapsulate the visual elements most commonly used in applications. Using these standard panels is usually the easiest way to integrate simple visual elements with your application's interactive human interface elements.

Icons

An icon is a graphic representation of some human interface element, such as a document, disk, folder, application, or the Trash in the Finder. The Finder draws and manages the icons that a user sees on the desktop.

Figure 1-16 shows the Note icon commonly used to identify Note alert boxes.

Figure 1-16 A Note icon panel

To display an icon, you typically use an icon panel. An icon panel encapsulates an icon and can draw itself appropriately within an embedding panel. Icon panels include methods for creating, drawing, labeling, and disposing of icons.

It's also possible to display an icon using the Icon Utilities, a lower-level set of utilities for manipulating icons that aren't inside panels. For example, you can use the Icon Utilities to display icons in your application's content area. The Icon Utilities also allow you to obtain the icon currently being used by a particular file in the Finder so you can display it.

The chapter "Icons," which will be available with later developer releases, describes in detail how to create and manipulate icons.

Pictures

To display a QuickDraw picture, you typically use a picture panel. Figure 1-17 shows an example.

Figure 1-17 A picture panel

To display a picture that's not in a panel, you can call QuickDraw directly.

Visual Separators

Visual separator panels can display horizontal, vertical, or rectangular visual separators. Figure 1-18 shows examples of horizontal and visual separators. A rectangular visual separator can optionally include a title. For an example of a rectangular visual separator panel, see Figure 1-7 on page 1-15.

Figure 1-18 Horizontal and vertical visual separator panels

To display visual separators that aren't part of a panel, you can use the Appearance Manager primitives shown in Table 1-2 on page 1-38.

Copy, Paste, Drag, and Drop

Users of Copland applications should be able to copy and paste data freely within the same window or from one window to another. Users should also be able to drag visual interface elements such as text, graphics, bitmaps, icons, or outline items from one place to another by pressing down on the mouse button while the pointer is over a selection, moving the pointer across the screen, and then releasing the mouse button.

You can use the Scrap Manager, Clipboard Manager, and Drag Manager to implement copy, paste, and drag and drop in a single piece of code. The Scrap Manager provides the generic transport package format, and the Clipboard and Drag Managers provide transport mechanisms for moving the packaged data, called a *scrap,* from one place to another.

Because they don't involve drawing to the screen, the Scrap Manager and the Clipboard Manager execute in a preemptively safe manner and are fully reentrant—unlike the Drag Manager and the rest of the Copland Toolbox, which are non-reentrant.

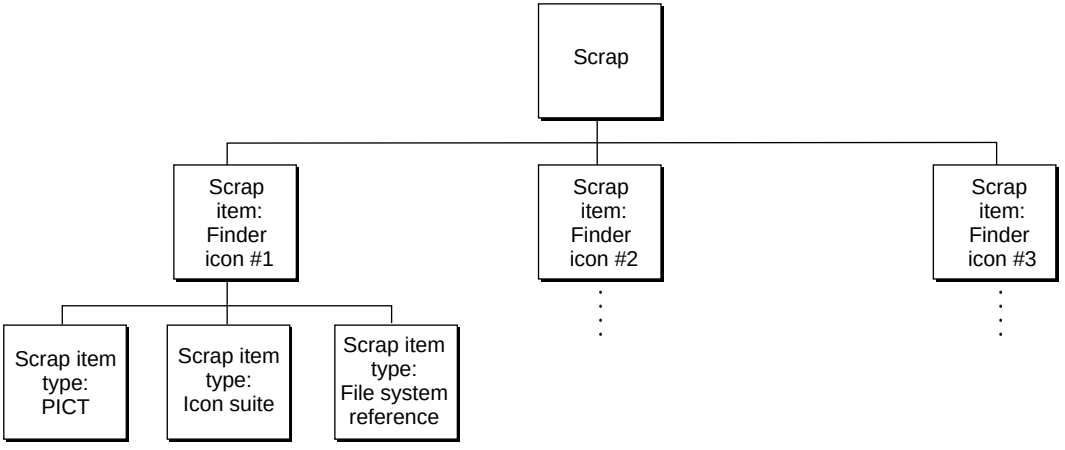
The chapter "Scrap, Clipboard, and Drag Managers," which will be available with later developer releases, describes in detail how to implement copy, paste, and drag and drop. The sections that follow introduce these three managers.

Scrap Manager

The Scrap Manager can handle data of any size, including QuickTime movies, sound data, graphics data, and other data that take up a lot of memory. The Scrap Manager provides functions that allow you to create a scrap, package the data to be transported inside it, and retrieve the data after the Clipboard Manager or Drag Manager have transported the scrap to its destination.

A **scrap** consists of one or more **scrap items.** Each scrap item is associated with a single piece of data represented by one or more **scrap item types.** For example, a scrap that contains a picture might contain a single scrap item represented by a single scrap item type, such as a PICT; whereas a scrap that contains a Finder icon might contain a scrap item represented by several scrap item types, including the icon itself, a PICT, and a reference used by the file system to identify the file associated with the icon, as shown in Figure 1-19. It is usually desirable to provide the same data in several different formats, so that the receiving application can choose a format that it can handle.

Figure 1-19 Scrap, scrap items, and scrap item types



The Scrap Manager also supports the concept of **promises.** For example, you can choose to put a scrap on the Clipboard as a promise instead of the actual data. This involves constructing an empty scrap with placeholders for the various scrap item types. When the user pastes, the Scrap Manager asks the original application to fulfill its promise for the type of data being pasted by providing the actual data. This mechanism avoids data transfer until the data is actually needed for a paste. It also allows the original application to transfer the data using just the format requested by the pasting application rather than duplicating the data in a variety of possible formats. Promises are especially useful for copying large pieces of data, but they are also the fastest way to copy any kind of data.

Promises placed on the Clipboard require slightly differently treatment than promises used in dragging operations. If the user quits the original application or closes the document containing the promised data before a promise on the Clipboard has been fulfilled, the original application should fulfill the promise before it quits or closes the window to ensure that user can paste the item in the future.

Clipboard Manager

The Clipboard Manager provides a mechanism for placing a scrap on the Clipboard and retrieving it when the user pastes the data in a new location. A single Clipboard is shared by all currently running applications.

In general, after you use the Clipboard Manager to place a scrap on the Clipboard, the scrap becomes read only. Multiple applications can use the Scrap Manager simultaneously to extract data from the scrap during separate paste operations, but the scrap can't be altered. If another scrap gets placed on the Clipboard before one or more applications have finished pasting, the Scrap Manager maintains the old scrap until they are finished, but treats the most recent scrap as the current scrap for any new paste operations.

Whenever a new scrap gets placed on the Clipboard, the Clipboard Manager sends an Apple event to all interested applications to notify them that the contents of the Clipboard have changed and what data types are available for the new data. This allows each application to update its Edit menu appropriately as soon as new data is copied to the Clipboard. For example, when your application receives an event informing it that the Clipboard now contains text in a format your application can handle, it should make sure that the Paste item in its Edit menu is enabled.

Drag Manager

From the user's point of view, to **drag** something means to position the pointer on a visual interface element (such as an icon in the Finder), press and hold the mouse button, move the pointer to a new position, and then release the mouse button. In general, dragging can have different effects, depending on what's under the pointer when the user first presses the mouse button. These can include selecting blocks of text, choosing a menu item, selecting a range of objects, shrinking or expanding an object, or moving an icon or other visual elements from one place to another. The Drag Manager supports the latter form of dragging: moving visual elements and their associated data from one place to another.

You use the Drag Manager to support the dragging of visual interface elements within your application, from your application to other applications or the Finder, and from other applications or the Finder to your application. The Drag Manager uses a scrap to hold the data associated with a dragged element. Elements that may be dragged can include text, graphics, bitmaps, icons, outline items, and so on. The Finder itself uses the Drag Manager to support

common dragging operations such as moving a file or folder or dropping items on other items to make something happen, such as running a script.

The Scrap Manager packs and unpacks the data that's transported in a drag. The Drag Manager supports the user experience while an item is being dragged, including displaying a transparent version of the original image during dragging. You use the Drag Manager to create the scrap associated with a dragged element, the Scrap Manager to add scrap items and their associated scrap item types to the scrap, the Drag Manager to handle the actual dragging, and the Scrap Manager to read the scrap after the drag operation is complete.

You must supply drag tracking and drag receive handlers that the Drag Manager uses to track the drag across the screen and receive the drag when the user drops drag items at a destination within your application.

Interactions With the Finder

Once you've designed your application, you need to create icons to represent the application and the documents it creates. The Finder displays these icons to the user. If your application appears as an item in the Apple or Application menu, your application's icon is displayed next to its name and, when your application is active, as the title of the Application menu.

Many applications allow users to set various preferences, such as default font, pen widths, menu contents, toolbar contents, backup saving behavior, and so on. The Preferences Manager provides a standard mechanism for controlling your application's preferences. Using the Preferences Manager ensures that your application can take advantage of Copland's support for multiple users who share a single computer.

The chapter "Finder Interface" describes how to define and create the icons for your application and its documents. The chapter also describes how your application interacts with the Finder. The chapter "Preferences Manager" describes how to implement user preferences for your application. Both chapters will be available with later developer releases.

Resources

Resources are basic elements of every Macintosh application. By defining descriptions of menus, windows, controls, dialog boxes, sounds, fonts, and icons in resources, you can make these and other elements easier to create and

manage. Using resources also eases translation of user interface elements into other languages.

A **resource** is any data stored according to a defined structure in the resource fork of a file. The data in a resource is interpreted according to its resource type. You usually create resources using a resource compiler or resource editor. This book shows resources in Rez format; Rez is a resource compiler provided with the Macintosh Programmer's Workshop (MPW), available from APDA. Apple and third parties also provide additional resource tools you can use to create the resources for your application.

Most of the Toolbox services use the Resource Manager to read resources for you. For example, you can use the Window Manager and panel methods to read descriptions of your application's windows, dialog boxes, menus, and controls from resources. The Toolbox services interpret a resource's data for you once it is read into memory. While you'll typically use Toolbox services to access resources, you can also use the Resource Manager directly to read and write resources.

The chapter "Resource Manager," which will be available with later developer releases, describes the Resource Manager in detail. To help you understand how the Window Manager and the Panels class library use resources, this section gives a brief overview of resources.

The Mac OS treats a **file** as a named, ordered sequence of bytes that is stored on a volume and is typically divided into two forks, the data fork and the resource fork. The **data fork** contains data that usually corresponds to data created by the user; the application creating the file can store and interpret the data in the data fork in whatever manner is appropriate. The **resource fork** of a file consists of the resources themselves.

When you write data to a file, you write to either the file's resource fork or its data fork. You must use File Manager routines to read from and write to a file's data fork and Resource Manager routines to read from and write to a file's resource fork.

You typically store as resources data that has a defined structure—such as icons and sounds—and descriptions of menus, controls, dialog boxes, and windows. When you create a resource, you assign it a resource type and resource ID. A **resource type** is a sequence of four characters that uniquely identifies a specific type of resource, and a **resource ID** identifies by number a specific resource of that type. (You can also use a resource name in place of a resource ID to identify a particular resource within a resource type.) For example, to create a description of a window in a resource, you create a resource of type `'wind'`

and give it a resource ID or resource name that is unique among any other `'wind'` resources that you have defined. Some resources have restrictions on the numbers you can use for resource IDs; in general, numbers 128 through 32767 are available for your use.

The Mac OS defines a number of standard resource types. You can use these resource types to define their corresponding elements. You can also create your own resource types if your application needs resources other than the standard types.

When your application or a Toolbox service requests a resource of a particular type with a given resource ID, the Resource Manager looks for the specified resource and, if successful, reads it into memory. However, the Resource Manager does not interpret the format of an individual resource type. You should not make any assumptions about a standard resource's format once the Resource Manager has read it into memory. For example, when you use the Window Manager to read a description of a window from a `'wind'` resource, the Window Manager uses the Resource Manager to read the resource into memory. Once the resource is in memory, the Window Manager interprets the resource's data and creates a window with the characteristics described by the resource. You should not directly access the window resource in memory. In general, the only resources you should access directly in memory are those whose formats you define yourself.

You typically store the resources specific to your application—such as descriptions of its menus, windows, controls, and dialog boxes—in the application file's resource fork. Whether you store data in the data fork or the resource fork of a document file depends largely on whether you can structure that data in a useful manner as a resource. Data that is likely to be edited by the user is usually stored in the data fork of a document file. Document-specific settings, such as the document window's last position and size on the screen, are usually stored as a resource in the document file's resource fork. The next time the user opens the document, your application can read the position and size saved in this resource and position the document accordingly.

You can specify that the Resource Manager read a resource into memory immediately when the Resource Manager opens a file's resource fork, or you can specify that the Resource Manager read it into memory only when needed. Normally, the Resource Manager stores resources from resource forks opened by your application in relocatable blocks in your application's heap. You can also specify whether the resource should be purged from memory to make room in memory for other data. If you specify that a resource is purgeable, you

need to use the Resource Manager to make sure the resource is in memory before accessing it.

When a user opens your application, your application's resource fork is opened automatically. When your application opens a file, your application typically opens both the file's data fork and the file's resource fork. When your application requests a resource from the Resource Manager, the Resource Manager follows a specific search order. (If necessary, your application can change the search order using Resource Manager routines.) The Resource Manager normally looks first for the resource in the resource fork of the last file that your application opened. So, if your application has a single file open, the Resource Manager looks first in that file's resource fork. If the Resource Manager doesn't find the resource there, it continues to search each resource fork open to your application in the reverse order that the files were opened. After looking in the resource forks of files your application has opened, the Resource Manager searches your application's resource fork. If it doesn't find the resource there, it searches system resources.

This search allows your application to use system resources, to override system resources with resources stored in the application's resource fork, and to override application-defined resources with resources stored in a document's resource fork.

A resource fork can contain at most 2700 resources. In general, you should not create more than 500 resources of the same type in any one resource fork.

Themes

As shown in the first two figures in this chapter, users can select different themes, or styles—that is, coordinated sets of human interface designs that determine the appearance of human interface elements on a systemwide basis, across multiple applications. Regardless of the theme, the core user experience remains the same, and users can switch themes without having to learn new human interface metaphors. The Appearance Manager and its use of **interface definition objects (IDOs),** which are extensible SOM-based definitions for the appearance of windows, menus, and controls, provide the underlying support for these capabilities.

IDOs are essentially drawing engines that use the inheritance characteristics of SOM to simplify the creation and customization of human interface elements. Unlike panels, they do not encapsulate data or track content in any way and do not need to be reinstantiated for every interface object. For example, the

Window Manager can use a single IDO to draw any number of identical windows.

A theme and its associated IDOs determine the appearance of all human interface elements on the screen, including alert icons, controls, background colors, dialog boxes, menus, screen savers, state transitions, and windows. Apple supplies several standard themes. Users can choose among the themes available to the system with the Appearance control panel, which also allows them to modify other aspects of their computing environment's appearance, such as the desktop pattern, highlight color, screen saver, and system font. (The Appearance control panel replaces the Desktop Patterns and Color control panels used in System 7.)

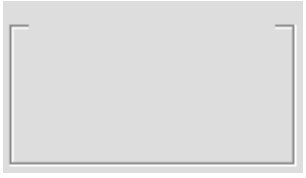
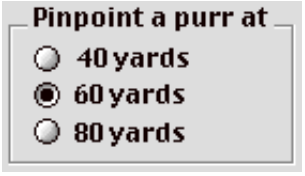
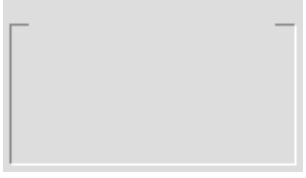
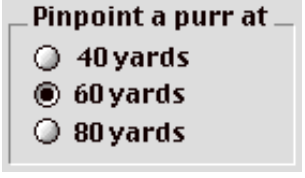
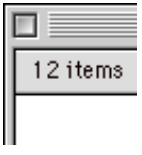
In addition to supporting user customization, themes and the underlying IDO mechanism insulate your application from future changes to the human interface. They free you from relying on hardwired appearances for standard elements while making it easier to create customized elements. Because Copland allows you to deal with appearance abstractions rather than specific details, your application can support not only the new human interface designs in Copland but also future design enhancements.

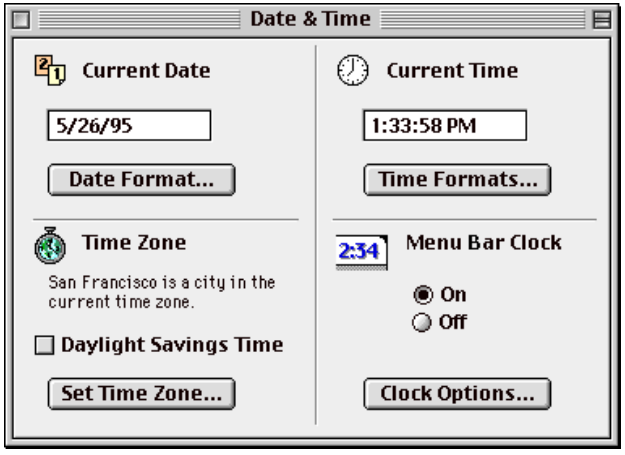
The Appearance Manager manages all aspects of themes and theme switching, including IDOs, the Appearance control panel, support for a variety of color data (RGB colors, pixel patterns, and so on), and support for animation and sound. It supersedes System 7 color tables such as `'cctb'` and `'mctb'` with a more abstract mechanism that allows you to coordinate colors with the current theme.

The Appearance Manager provides primitives for specifying window headers, group boxes, separators, and other building blocks that you can use to assemble custom, theme-compatible visual elements for specialized purposes. Table 1-2 shows preliminary designs for some of the primitives provided by the Appearance Manager as they appear in the Apple Default theme.

Using the Copland Toolbox as described in this book ensures that your application will support themes and theme switching.

Table 1-2 Some Appearance Manager primitives and examples of their use

Appearance in the Apple Default Theme	Example of use	Description
		Primary group box. Used to frame a primary group of related controls; title optional.
		Secondary group box. Used to frame a secondary group of related controls; title optional.
		Placard. Used as background for an element in a window.
		Window header. Used to display information at the top of the content area, below the title bar.

Appearance in the Apple Default Theme	Example of use	Description
		Vertical and horizontal separators. Used to separate elements in a dialog box or window.

In addition to the primitives illustrated in Table 1-2, the Appearance Manager provides functions that allow you to determine how the current theme is drawing various aspects of the human interface, such as background fills.

You need to use the Appearance Manager directly only if you're using the primitives illustrated in Table 1-2 to create custom human interface elements or if you want to draw in your application's content area in a manner that matches the current theme. For example, you can ask the Appearance Manager for the current background color so you can coordinate the appearance of your application's content area with the current theme. Similarly, if you want to draw a line through a standard menu item, you can ask the Appearance Manager for the current color of the menu item text so you can use the same color for the line.

The Panels class library uses Appearance Manager primitives to create some kinds of panels. For example, a visual separator panel uses the primary group box or vertical or horizontal separators. If you want an element to exist cooperatively with other panels, instantiating and drawing itself as appropriate, you should use the Panels class library to create it. If you want to

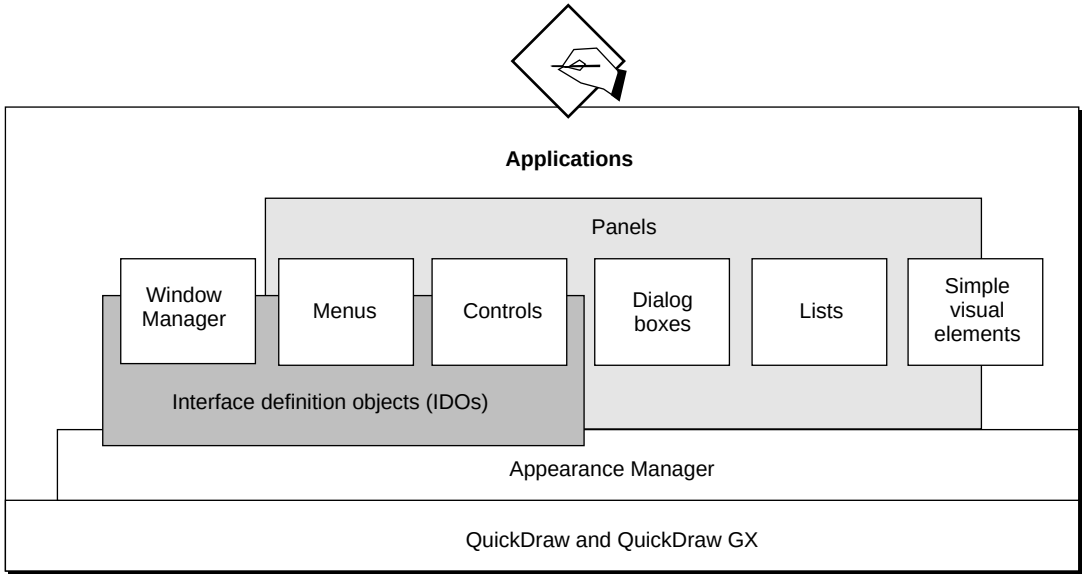
exercise complete control over the element you are creating—for example, in a content area that doesn't use panels at all—you should use the primitives directly.

The chapter "Appearance Manager," which will be available with later developer releases, describes the Appearance Manager in detail.

Copland Toolbox Architecture

Figure 1-20 illustrates the relationships among the Copland Toolbox services directly involved in creating an application's human interface.

Figure 1-20 High-level architecture of the Copland Toolbox



The parts of the Toolbox shown in Figure 1-20 play the following roles:

- The Window Manager creates and manipulates windows.
- The Panels class library provides a uniform, object-oriented interface for implementing keyboard navigation, mouse interaction, copy, paste, drag and drop, and other standard behavior for menus, controls, dialog boxes, alert boxes, lists, and simple visual elements. Panels use the Appearance Manager, menu IDOs, and control IDOs to draw themselves. You use SOM techniques to create and manipulate panels.
- The Appearance Manager keeps track of all aspects of themes and theme switching, including window IDOs, menu IDOs, control IDOs, and other information specified by the current theme.

The Toolbox also includes several services not shown in Figure 1-20:

- The Clipboard Manager and Drag Manager use the Scrap Manager to transport data by means of copy, paste, and drag and drop operations.
- The Scrap Manager provides a generic transport package format used by the Clipboard Manager and Drag Manager.
- The Finder interface lets you specify icons that represent your application and its documents in the Finder.
- The Preferences Manager provides a standard mechanism for controlling your application's preferences.
- The Resource Manager manages the resources in which you specify your application's human interface elements

IDOs provided by the current theme determine the appearance of human interface elements on the screen with the aid of the Appearance Manager, QuickDraw, and QuickDraw GX. For example, when your application asks the Window Manager to draw a window, it normally uses the window IDO from the current theme to do the actual drawing. Similarly, when your application calls a panel's `Draw` method, the panel uses the menu IDO, control IDO, or Appearance Manager primitives specified by the current theme to draw itself.

You can also use the Appearance Manager, QuickDraw, or QuickDraw GX to assemble your own human interface elements and content areas, bypassing the higher-level Toolbox services altogether.

All Toolbox services support similar capabilities in similar ways, thus ensuring a consistent programming interface as well as a consistent user experience. The most important architectural principles fall into four categories:

- **Opacity and consistency.** The Toolbox provides a complete programming model that doesn't require direct manipulation of underlying data structures.
- **Integrated support for international text.** The Toolbox takes advantage of Copland text objects to provide flexible support for multilingual text throughout the human interface.
- **Data extensibility.** The Toolbox uses the Collection Manager to support the addition of arbitrary tagged data to Toolbox data structures without manipulating the structures directly.
- **Design extensibility.** The Toolbox includes a comprehensive set of standard windows, menus, controls, and other human interface elements that can be used as is or extended by developers to support specialized application needs.

Opacity and Consistency

The Copland Toolbox provides high-level interfaces that eliminate the need to keep track of the internal organization of system data structures. Instead, Toolbox services ensure the opacity and consistency of the programming interface. The Toolbox provides

- accessor functions or methods for getting and setting information about specific human interface elements
- blind references such as `WindowRef` that identify data structures without permitting direct access
- high-level interfaces for all operations, including those traditionally associated with low-memory globals
- the Panels class library for creating standard, customizable human interface elements

By ensuring the opacity of the Toolbox interfaces, Apple can continue developing human interface capabilities without forcing applications to deal with new implementation details.

For an introduction to the use of accessors, blind references, the Panels class library, and other Toolbox interfaces, see the chapter "Introduction to Toolbox Programming," which will be available with later developer releases.

International Text

Copland supports a systemwide text data type, called a **text object,** that encapsulates the details of text encoding. Text objects allow applications to manipulate multilingual text transparently without dealing with the details of character encoding, which can be based on Unicode, ASCII, traditional Macintosh, and other encoding systems. Copland applications should use text objects rather than Pascal and C strings within all human interface elements, including menu item text, buttons, and window titles.

Copland's pervasive support for text objects has two ramifications for application programming:

- You can display multilingual text (in multiple scripts) within a single menu, dialog box, or other human interface element.
- Because you don't have to keep track of the details of individual scripts and encoding systems, localization of interface elements is greatly simplified.

In addition to supporting text objects throughout the Toolbox, Copland provides standard human interface elements, such as left-growing windows and automatically resizable dialog boxes, that support specific international needs.

For an introduction to the use of text objects with the Toolbox, see the chapter "Introduction to Toolbox Programming," which will be available with later developer releases.

Extensible Data Structures

The Copland Toolbox eliminates the need for hardwired modification of system data structures by providing a new mechanism that lets you attach arbitrary data to virtually any human interface element. Based on the Collection Manager, which originally shipped with QuickDraw GX, this mechanism can be used to associate data with a tag and ID, attach that data to any Toolbox data structure, and retrieve it when necessary.

The Window Manager and all panels provide collection item functions or methods you can use to get, set, and remove collection items associated with virtually any human interface element. Collection items can be used for a variety of purposes. For example, in a Preferences dialog box that allows the user to switch among several preference "pages," each of which displays multiple panels, you can use collection items to associate a page ID with the

panels that appear in that page. This makes it easy to hide or show the appropriate panels when the user switches pages.

The Collection Manager is described in *Inside Macintosh: QuickDraw GX: Environment and Utilities.* For an introduction to the use of collections with the Toolbox, see the chapter "Introduction to Toolbox Programming," which will be available with later developer releases.

Extensible Designs

Whenever possible, you should use the standard windows, menus, controls, and so on provided by the Copland Toolbox. This is the easiest way to support themes. If you need to create custom elements, you have two choices:

- **Customize standard elements.** You can modify just those characteristics of a standard element that you wish to implement differently while maintaining support for theme switching.
- **Implement your own theme-compatible elements.** You can assemble custom human interface elements from primitives and fills that maintain support for theme switching.

Copland provides two mechanisms for extending human interface designs: panels and IDOs. Both allow you to customize the standard designs or create new ones without sacrificing compatibility.

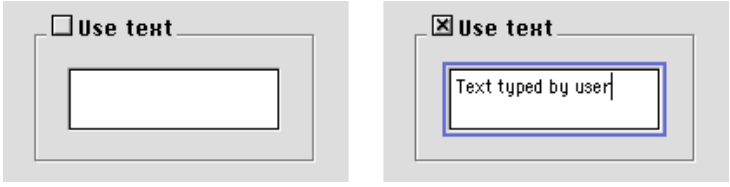
Customizing Panels

Figure 1-6 on page 1-13 shows a portion of the inheritance hierarchy for the standard Copland panels. You can use standard SOM techniques to subclass custom panels from any of the standard panel classes.

For example, you can create a custom text panel, based on the editable text panel class, that accepts numbers but not letters. To do so, you subclass from the editable text panel class and override just three methods: `Keydown`, `Paste`, and `Drop`.

Figure 1-21 shows another example of a custom panel, this one based on the embedding panel class.

Figure 1-21 Custom panel created by embedding a checkbox panel, rectangular visual separator panel, and editable text panel in an embedding panel.



Inactive Active with keyboard focus

The panel shown in Figure 1-21 includes an editable text box, a rectangular visual separator around the text box, and a checkbox at the top of the visual separator that allows the user to enable or disable the text box. To create this panel, you subclass from the editable text panel class to create an embedding panel that contains the other panels. You must override the embedding panel's `DoClick` method, calling the inherited `DoClick` to determine whether the click was in the checkbox and enabling or disabling the editable text panel.

The chapter "Panels," which will be available with later developer releases, describes how to assemble custom panels from the standard panels classes.

Customizing Interface Definition Objects

IDOs determine all aspects of the appearance of their respective human interface elements. For example, the Window Manager can use a single IDO to draw any number of identical windows. When it needs to perform operations such as drawing the window frame, resizing the window, reporting the region where mouse-down events occur, calculating content regions, and so on, it uses standard methods defined for that kind of IDO. This arrangement allows window operations to be entirely independent of the visual characteristics of a window defined by a particular IDO.

You can use the IDO mechanism to create custom IDOs in two ways:

- Subclass from the IDO you want to modify and override only those methods you want to change.
- Create your own custom IDO from scratch.

The chapter "Interface Definition Objects," which will be available with later developer releases, describes both ways of creating a custom IDO.

The Toolbox: System 7 Compared With Copland

Contents

This chapter compares some of the System 7 Toolbox services with the equivalent Copland services, including information on backward compatibility with System 7 applications and requirements for Copland-savvy applications. Where possible, this chapter categorizes Toolbox functions according to the categories used in the reference sections in *Inside Macintosh: Macintosh Toolbox Essentials.*

This chapter doesn't describe the interfaces that are entirely new in Copland, such as the Appearance Manager, Panels class library, and IDO class library. For a conceptual overview of the entire Copland Toolbox, see Chapter 1, "Introduction to the Copland Toolbox."

If you have developed a System 7 application and want to begin planning its migration to Copland, this chapter provides preliminary information to help you get started. However, it doesn't include detailed information about the Copland event model, panel methods, resource formats, human interface guidelines, and other aspects of Copland that you will need to understand before you can create a Copland-savvy application. Later developer releases will provide this information.

▲ **WARNING**

The interface descriptions in this chapter are preliminary and incomplete. All interfaces are subject to change in later developer releases.

In particular, the interfaces in this release for the Dialog Manager, Control Manager, List Manager, and Menu Manager will not be supported in later releases, and the Panels class library will be extended to provide the equivalent services. ▲

General Compatibility Guidelines

Using the Copland Toolbox not only ensures compatibility with Copland but also lays the foundation for new capabilities that will be introduced in Gershwin and future Mac OS enhancements.

Like any major system software revision, Copland introduces features that aren't backward compatible with earlier systems. However, most System 7 applications can run on Copland, even though they may not be able to take advantage of all its features. For example, clients of standard System 7 definition procedures (defprocs) work correctly and inherit the Copland human interface appearance. Custom defprocs written for System 7 also work correctly on Copland but do not inherit the Copland appearance.

Here are some guidelines you can use now to ensure that System 7 applications currently under development are compatible with the Copland Toolbox:

- Support Apple events as described in *Inside Macintosh: Interapplication Communication*, including factoring your application and making it fully scriptable and recordable. The Copland event model is based primarily on Apple events.
- Don't assume that dialog box backgrounds are white. The Copland human interface supports a variety of background colors.
- For floating windows, use the standard floating window definition (ID 124) introduced in System 7.5. This window definition works correctly on Copland and inherits the Copland appearance.
- Don't hard-code any assumptions about the precise locations of human interface elements such as close boxes, zoom boxes, and window titles within the noncontent areas of windows or dialog boxes.
- Don't hard-code any assumptions about the precise locations of any human interface elements in the Save and Open dialog boxes. Use the relative position of the standard elements to determine the locations of new ones.
- Never access low memory directly. If you need to access low memory, use accessor functions.

- Use data structure accessor functions where they exist. For example, use `SetMenuItemText` and `GetMenuItemText` to manipulate menu item text rather than accessing the data structure directly.
- If data structure accessor functions aren't available, isolate the code that accesses data structures directly. Copland provides accessor functions for all data structures, and it is easier to take advantage of them if you have isolated the code that needs to be updated.
- Don't manipulate the window list directly. Use the `BringToFront` and `SendBehind` functions instead.

The Copland Event Model

Chapter 1, "Introduction to the Copland Toolbox," introduces the Copland event model. Copland supports the classic Event Manager as described in *Inside Macintosh: Macintosh Toolbox Essentials* for backward compatibility.

When launched, a Copland-savvy application calls `AEInstall` to install handlers for the events that it wants to handle. It doesn't use `WaitNextEvent` to receive events and `AEProcessAppleEvent` to dispatch them. Instead, the application uses the new Copland function `AEReceive`, which processes events with other Apple event consumers in the preemptively scheduled Copland environment.

A Copland application consists of one or more tasks. Copland tasks are introduced in the accompanying document, "Kernel and Operating System Services." Typically, the code for a task performs any initialization work the task requires and then calls `AEReceive`, which doesn't return unless there's an error. For the most part, the task executes from inside `AEReceive`, which takes care of many of the operations that are handled by a System 7 event loop. The `AEReceive` function blocks the calling task until an event the application can handle arrives, then reawakens the task and dispatches the event to the appropriate handler. This cycle of blocking then waking the task continues until the task terminates or the application quits.

All events are conveyed via the "bottleneck" of `AEReceive`. This avoids the multilevel dispatching required with the classic Event Manager, which recognizes three principal kinds of events (low-level events, operating system events, and high-level events such as Apple events), each requiring different treatment.

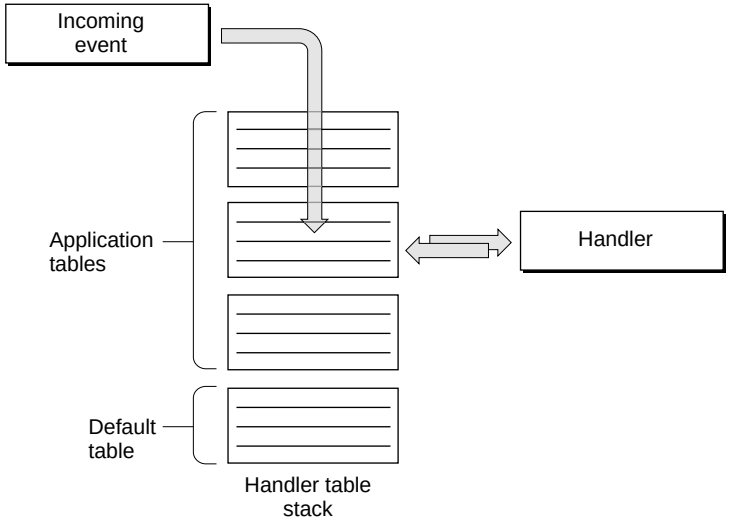
The sections that follow introduce some of the concepts underlying the Copland event model. Later developer releases will provide more detailed information about handling events in Copland.

Apple Event Dispatchers

The ultimate target of an event is always an **Apple event dispatcher,** which combines an event queue, at least one task, and a stack of handler tables. Every process has a default Apple event dispatcher, and your application may install additional dispatchers as necessary.

Figure 2-1 illustrates the way an Apple event dispatcher uses its handler tables to dispatch an event.

Figure 2-1 A handler table stack associated with an Apple event dispatcher



When an event arrives, `AEReceive` wakes up the task and searches the stack of handler tables for a matching handler. An Apple event handler is identified by the combination of the event class and event ID of the event it handles—much the way Apple event handlers are stored in application and system handler tables in System 7. The main difference is that Copland has no handler tables

comparable to the global handler tables in System 7. Each Apple event dispatcher maintains a separate handler table stack, which consists of a default handler table and one or more application handler tables.

The **default handler table,** which is identical for all applications and cannot be modified, provides a set of default event handlers. Most applications need to install one or more of their own handler tables to augment the default behaviors.

Application handler tables are installed by your application. You can use Apple Event Manager functions to add, remove, or replace application handler tables at any time.

You can stack application handler tables on top of the default one to express interest in events and intercept them as the Apple Event Manager matches incoming events with handlers. When the Apple Event Manager searches for an event's handler, it starts from the top of the stack and looks down the chain of handler tables until it finds a match.

Apple Event Handlers

When the Apple Event Manager locates an event's handler in a handler table, it passes the event to that handler. The handler must make several decisions about how to handle each event:

1. **Handle the event or don't handle it.** If it actually handles the event, the handler must then choose whether to filter it. If the handler doesn't handle the event, it allows the event to "fall through" to the next table in the stack, and the Apple Event Manager keeps searching for a handler.
2. **Filter the event or don't filter it.** If it filters the event, the handler suspends the event temporarily until there is a change in modality. For example, a handler might filter certain events while a dialog box is being displayed, allowing those events to be handled only after the user has dismissed the dialog box. If it doesn't filter the event, the handler must respond to the event in some way and then decide whether to pass the event on to the next table in the handler table stack.
3. **After handling the event, pass it on or not.** After handling the event, the handler can choose to let it fall through to next table in the stack, in which case the Apple Event Manager keeps searching for a handler.

Events for which handlers can be installed in a handler table include the following:

- **Physical events** are low-level events—such as disk inserts, update and activate events, mouse clicks, and keypresses—that are generated by the system. Most of the handlers provided by the default handler table are physical event handlers. Physical event handlers translate physical events into higher-level synthetic events and send the repackaged events to the same Apple event dispatcher. For example, the Text Services Manager can translate a series of keypresses into a single Kanji character.
- **Synthetic events** are events that are meaningful to most applications, such as "window zoomed" or "menu selected." More than one physical event may generate the same synthetic event; for example, the user might select a menu item by releasing the mouse button or by pressing a key. The default handler table provides some synthetic event handlers that basically redirect the event to a particular menu or window. Synthetic event handlers can also translate synthetic events into higher-level semantic events.
- **Semantic events** are events such as "Open Document" or "Quit Application" with a specific meaning for an individual application. Semantic events can be generated by synthetic events; for example, the synthetic event handler for the Quit command in the File menu in turn generates a Quit Application semantic event. More commonly, semantic events are injected directly into the corresponding handler rather than cascading up through synthetic events from an original physical event. For example, a Get Data event is always generated by a script or another application; it is never generated directly by the user.

When you're creating the human interface for your application, you are primarily concerned with synthetic events. Most synthetic events are ultimately directed at a target within the application, such as a window or a menu. Combined with the Panels class library, the event-handling mechanism provided by Apple event dispatchers allows the Toolbox to help your application arbitrate targets for some events while allowing you to override the default arbitration at any point.

Semantic events are primarily of interest to external clients. Unlike other kinds of handlers, semantic event handlers can send replies in response to a script or to a query from another application.

Tasking Models

A Copland application has a single primary task and may have additional secondary tasks. Only the primary task can call Toolbox services. Tasks are discussed in more detail in the accompanying document, "Kernel and Operating System Services."

When any task calls `AEReceive`, the task specifies the Apple event dispatcher (and thus the event queue) in which it's interested. You can associate tasks with Apple event dispatchers in three principal ways:

- one task and one dispatcher
- multiple tasks and multiple dispatchers
- multiple tasks and one dispatcher

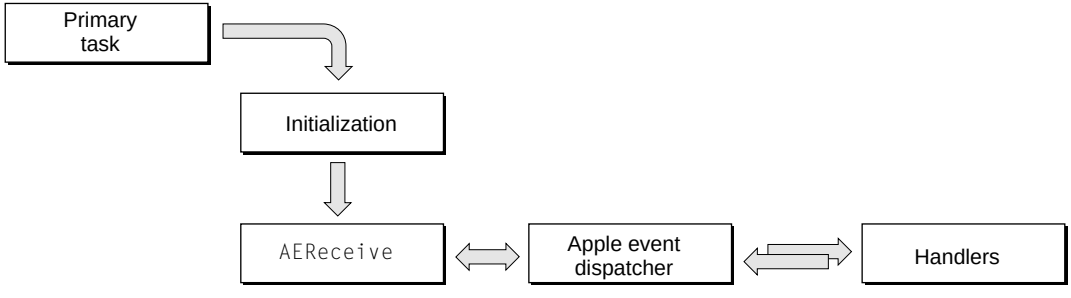
The sections that follow introduce these tasking models. More complex relationships among tasks and dispatchers may be created by combining these approaches in various ways.

Identifying the particular arrangement of tasks and dispatchers appropriate for your application is a design decision. In general, a single primary task should be associated with a single Apple event dispatcher, as in the first two models. The third model, which associates multiple tasks with a single dispatcher, is intended for use by server processes.

One Task, One Dispatcher

Figure 2-2 shows the simplest case: A single primary task associated with a single Apple event dispatcher.

Figure 2-2 One task, one dispatcher



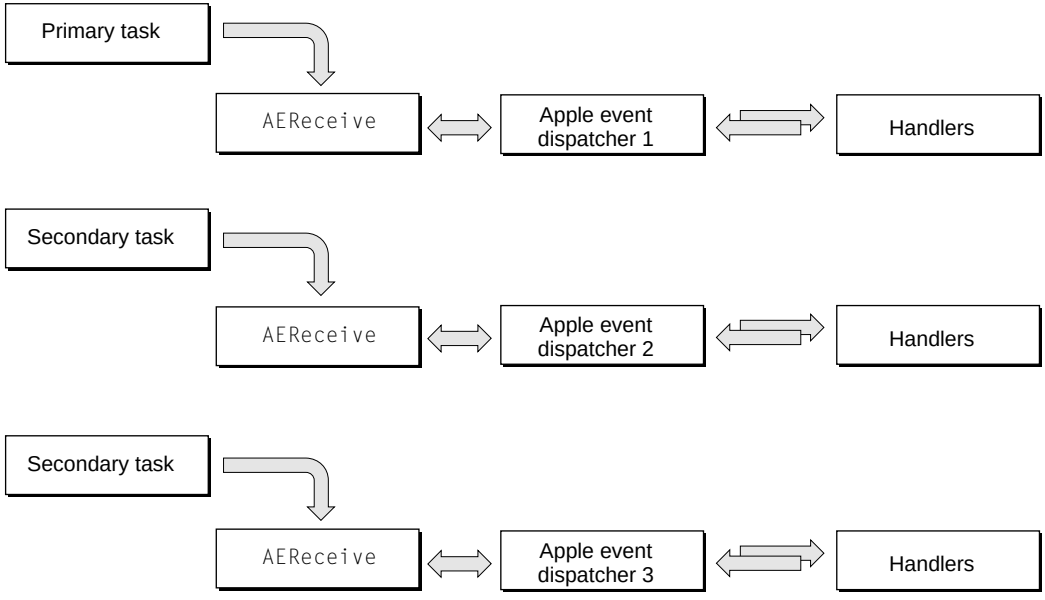
The arrangement shown in Figure 2-2 superficially resembles a System 7 event loop in the sense that a single task is responsible for all event handling. The `AEReceive` function wakes the application's primary task each time an event arrives and uses the application's single Apple event dispatcher to locate the event's handler.

Multiple Tasks, Multiple Dispatchers

An Apple event dispatcher and its handlers represent one kind of behavior or set of activities that your application can perform. For example, it makes sense to associate an application's primary task with a single dispatcher for all handlers that call Toolbox functions, as in Figure 2-2.

It's also possible to create additional dispatchers for one or more secondary tasks, which cannot call Toolbox functions. Figure 2-3 illustrates this arrangement.

Figure 2-3 Multiple tasks, multiple dispatchers



You can implement the tasking model shown in Figure 2-3 in several ways. It's possible, for example, to route events to a particular dispatcher. You can route all human interface events to the dispatcher associated with the primary task, and other events to dispatchers associated with secondary tasks as appropriate.

Alternatively, you can send all events to the primary task initially and use the primary task's dispatcher to forward certain events to one or more secondary tasks. For example, a graphics program might have a menu command that transforms an image in some way by performing a series of calculations. The handler invoked by that command can in turn send an Apple event to a different dispatcher associated with a secondary task that actually performs the calculations. The primary task is then free to continue responding to the user's manipulation of the human interface while the secondary task, which doesn't involve the human interface, continues to execute in the background.

When the secondary task needs to inform the user of its progress, the handler that's performing the calculation can send an event back to the primary task's dispatcher to update a progress indicator. Similarly, when the handler has completed its calculations, it can send an event back to the primary task's dispatcher to invoke the handler that actually draws the transformed image.

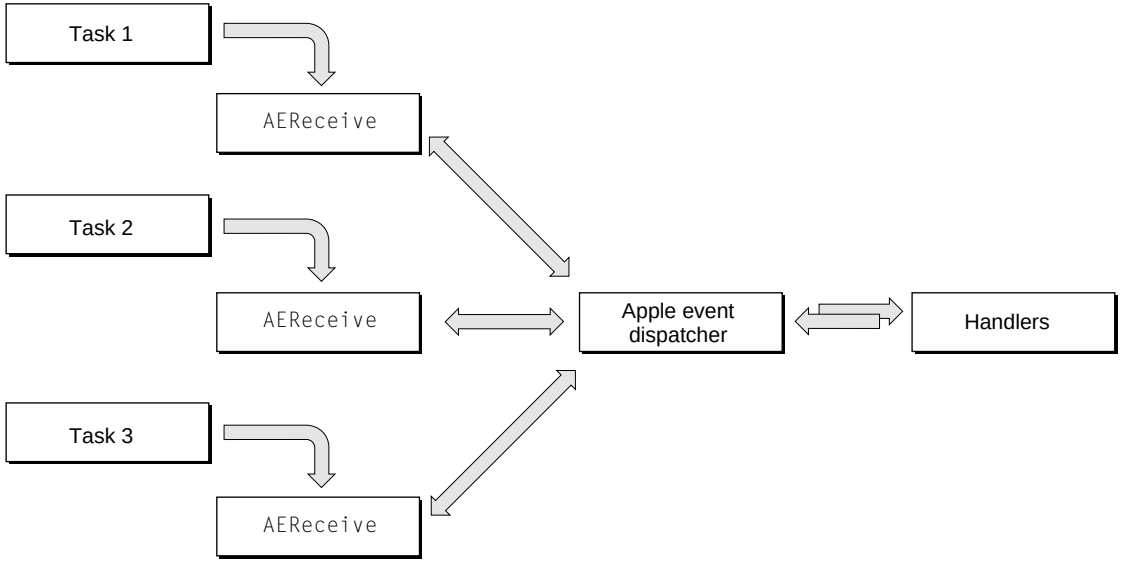
Because Copland permits an application to use multiple secondary tasks in addition to a single primary task, the graphics application in this example could actually perform transformation calculations on several different images, starting each calculation at a different time and performing them all concurrently. Thus, the primary task could be drawing the results of one calculation to the screen while one secondary task is in the middle of calculating a transformation for a second image and another secondary task is just beginning to calculate a transformation for a third image.

Because of the way the Copland kernel prioritizes tasks in this kind of situation, the application continues to be highly responsive to user actions even while secondary tasks are executing—unlike System 7, in which background processing can seriously interfere with the application's responsiveness.

Multiple Tasks, One Dispatcher

Figure 2-4 shows multiple tasks calling `AEReceive` with the same dispatcher. Each task has its own entry point and begins executing at a different time. The tasks don't necessarily have to be identical, but they must use the same set of handlers provided by the dispatcher and must all be equally qualified to deal with incoming events. All handlers in a dispatcher that is shared in this way must be fully reentrant.

Figure 2-4 Multiple tasks, one dispatcher



This arrangement is most useful for server applications. For example, a database that receives requests continuously from several sources can spawn a series of identical tasks associated with the same Apple event dispatcher. All these tasks share the same stack of handler tables. The Apple event dispatcher pairs each task with each incoming request and looks up the corresponding handler in the stack of handler tables. As each task resumes execution, it can execute at the same time, if necessary, that previously woken tasks are executing. Thus, the database can handle a series of requests simultaneously.

The tasking model shown in Figure 2-4 is not appropriate for most client applications or for applications that interact directly with the user.

Benefits of the Copland Event Model

You must support the Copland event model to take advantage of all the human interface features provided by the Copland Toolbox. The Copland event model also provides these benefits:

- The use of blocking rather than polling improves performance for all applications running on the same machine and takes maximum advantage of priority-based preemptive scheduling.
- The use of Apple event handlers rather than event masks to distinguish events permits a much larger name space for events. This ensures that Apple can provide new default handlers and new behaviors with minimum impact on existing applications and also makes it easier to create specialized events for your own purposes.
- The use of Apple events throughout the system ensures that all Copland-savvy applications can be scriptable and recordable.
- Events are always sent and dispatched the same way, which simplifies the overall Mac OS programming model.

Although Copland supports the classic Event Manager for backward compatibility, many new Toolbox features require the new event model, and much of the information conveyed by Copland Apple events is lost in the translation to classic events.

More information about the Copland event model will be available with later developer releases. The best way to prepare your System 7 application for Copland events is to support Apple events as described in *Inside Macintosh: Interapplication Communication*, including factoring your application and making it fully scriptable and recordable. In general, the new event interfaces provided by Copland are designed to augment the capabilities described in that book rather than to replace them.

Window Manager

Constants and Data Types

Window References

All Window Manager functions that take or return pointers of type `WindowPtr` or `WindowPeek` in System 7 use window references of type `WindowRef` in Copland. The `WindowRecord` data structure and other related structures that underlie window references are not directly accessible to Copland-savvy applications. Instead, you pass window references to accessor functions that get and set window characteristics (see "Accessing Windows" on page 2-17 for more details).

Window Classes

The Copland Window Manager defines a new `WindowClass` data type that determines a window's layering. When you create a window, you specify its class as normal, floating, or modal. Modal windows always appear above floating windows, which always appear above normal (document) windows. Once a window has been created, its class can't be changed.

When a modal window becomes visible, the Window Manager deactivates all the application's floating windows, generates deactivate events for them, and the menu bar changes to a modal state. The Window Manager reverses this process when it hides any modal window.

Window IDOs

The Copland Window Manager replaces the window definition functions (defprocs) used in System 7 with SOM-based window interface definition objects (window IDOs). For more information about IDOs, see Chapter 1, "Introduction to the Copland Toolbox."

Window IDOs support several new capabilities that aren't supported by System 7 definition functions. For example, a window IDO specifies a grow direction, which can be either left to right or right to left, and allows you to

locate specific parts of a window, such as the title text and the window's draggable region.

Window Attributes

In System 7, a window's visual and functional attributes are defined by a window definition ID that incorporates both the resource ID of a window definition function and a variant specifying structural differences between windows that are otherwise identical.

The Copland Window Manager replaces window defprocs with window IDOs and retains the variant mechanism for specifying basic structural differences such as modal dialog boxes, movable modal dialog boxes, and floating windows with a drag bar at the side instead of the top. In addition, the Copland Window Manager defines a new `WindowAttributes` data type that specifies additional details for a window, such as whether it has a grow box, collapse box, title bar icon, or right-to-left orientation; whether its content can be erased; and whether it should receive update or activate events.

Window Manager Functions

Initializing the Window Manager

This release replaces the System 7 function `InitWindows` with `InitWindowsVersion`, which takes the version number of the current Window Manager as a parameter. Although this release requires Copland-savvy applications to use `InitWindowsVersion`, this requirement is likely to change in later releases.

Creating Windows

The Copland Window Manager provides two new functions that create windows and return window references. The `GetNewWindowRef` function creates a new window on the basis of a description in a window resource. The `NewWindowRef` function creates a window on the basis of window attributes and other characteristics specified in its parameters. Copland-savvy applications should use these functions rather than `GetNewCWindow`, `GetNewWindow`, `NewCWindow`, and `NewWindow`, which are supported for backward compatibility only.

The System 7 Window Manager creation functions use window definition IDs to identify a window definition function and variant code, which determine how to draw the window. The Copland Window Manager creation functions use window IDOs rather than window defprocs and use variant codes only to identify broad structural differences among modal, movable modal, and floating windows. You can obtain a reference to the current theme's window IDO by using the Appearance Manager routine `GetSystemIDO`.

Neither `GetNewWindowRef` nor `NewWindowRef` include a parameter for storage of the window itself. The Copland Window Manager keeps track of window memory for you.

It is usually easier to use `GetNewWindowRef` to load a window from a previously defined resource rather than using `NewWindowRef` to create it dynamically. The `GetNewWindowRef` function uses a window IDO and a variant code to create a window according to a description in a window resource. Although an IDO and variant code can be specified in the window resource, you can also pass `GetNewWindowRef` a `UniversalIDOReference` structure that specifies a different window IDO and variant code.

A `UniversalIDOReference` is a structure defined by the Appearance Manager. It includes a pointer to an IDO and a variant code for the desired window structure. To override the IDO and variant code specified in the window resource, you pass `GetNewWindowRef` a pointer to a `UniversalIDOReference` structure. To use the window IDO and variant code specified in the window resource, you pass a null pointer instead of a pointer to a `UniversalIDOReference` structure.

Differences between `NewWindowRef` and its System 7 counterparts include the following:

- There are no title, `refCon`, or visibility parameters. A window is always invisible when it is created. You use other Window Manager functions to make a window visible, set its title, and keep track of your associated data.
- Instead of passing a window definition ID, you pass `NewWindowRef` a pointer to a window IDO and a variant code.
- You must specify the window's class (see "Window Classes" on page 2-14).

Naming Windows

When you use `GetNewWindowRef` to create a window, the name of the window is specified in a text object resource referred to by the window resource. To get or

set the title of a window after you have created it, you should use the Copland accessor functions `SetWindowTitle` and `GetWindowTitle`. Although the Copland Window Manager supports the System 7 functions `SetWTitle` and `GetWTitle` for backward compatibility, Copland-savvy applications should use the new functions to take advantage of Copland text objects.

Accessing Windows

The Copland Window Manager doesn't allow direct access to the data structures that underlie window references. Instead, you pass window references to new accessor functions that get and set a variety of window characteristics, including its attributes, rectangle, region, kind, visibility, highlighting, and title. You can use additional accessors to get characteristics that can't be set, such as the structure region, content region, update region, class, reference constant, and window IDO.

The new accessor functions include a new convenience function, `GetWindowPort`, that provides a way to get a window's graphics port. You must use this function to get a window's port; you cannot assume that a window reference is the same as a graphics port.

Manipulating Window Collection Items

The Copland Window Manager provides three functions for manipulating the collection items associated with a window: `AddWindowCollectionItem`, `GetWindowCollectionItem`, and `RemoveWindowCollectionItem`. You can use these functions to access collection items defined by the Window Manager or to add and remove your own collection items.

The Window Manager defines collection item tags for the following window data:

- handle to the title bar icon suite
- handle to the custom title bar gadget icon suite
- handle to the window content pattern
- RGB color for the window content area

If collection items for a particular window define both a pattern and an RGB color, the Window Manager ignores the color and uses the pattern.

Displaying Windows

The `ShowWindow`, `HideWindow`, `ShowHide` (renamed `ShowHideWindow`), `HiliteWindow`, and `DrawGrowIcon` work much the same way in Copland as they do in System 7.

Manipulating Window Layering

In the Copland Window Manager, the `SelectWindow`, `BringToFront`, and `SendBehind` functions move a window only within that portion of the application's layer devoted to the window's class. For example, calling `SelectWindow` on a document window brings it to the front of all document windows, but not in front of modal or floating windows. Similarly, `SendBehind` doesn't permit invalid ordering of windows. Applications that aren't Copland-savvy aren't affected by this change because they don't use the new floating and modal window classes.

Copland-savvy applications can continue to use the `FrontWindow` function to get the frontmost visible window, regardless of its class. The Copland Window Manager also provides several new functions that act on class-specific portions of the window list in an application's layer. These include the following:

- `GetWindowList` returns the frontmost window. The semantics of this function will change in future releases.
- `FrontNonFloatingWindow` returns the frontmost modal or document window.
- `FrontWindowOfClass` returns the frontmost window of a given class.
- `AreFloatersVisible` returns the visible state of all floating windows.
- `ShowHideFloatingWindows` sets the visible state of all floating windows.
- `ActivateFloatingWindows` activates or deactivates all floating windows.

Positioning Windows

The Copland Window Manager provides three new functions—`AutoPositionWindow`, `PositionWindow`, and `CheckWindow`—that allow you to position windows or get information related to window positioning by passing constants for various control values. These calls greatly simplify window-positioning tasks that you must code explicitly in System 7. For example, `AutoPositionWindow` automatically takes care of staggering multiple windows so they don't overlap.

Retrieving Window Information

Current plans for the Copland version of the `FindWindow` function include several new result codes:

- The `inCollapseBox` constant indicates that a mouse-down event has occurred in the window's collapse box. Copland-savvy applications should respond by calling the `CollapseWindow` function.
- The `inTitleIcon` constant indicates that a mouse-down event has occurred in the window's title bar icon. Copland-savvy applications should respond as follows: Use the Drag Manager to create a drag scrap reference, pass that reference to the Window Manager function `BeginTitleIconDrag` to highlight the icon, call the Drag Manager to handle the actual dragging and dropping, then call `EndTitleIconDrag`.
- The `inCustomGadget` constant indicates that a mouse-down event has occurred in the window's custom gadget. A custom gadget is an application-defined element in the title bar.

Note
The Copland event model and the Panels class library make it possible to manipulate windows without calling the `FindWindow` function directly. Later developer releases will include more information on this topic. ◆

Unlike the System 7 Window Manager, the Copland Window Manager won't attempt to activate the palettes of floating windows. Copland-savvy applications either can rely on the Window Manager to activate windows appropriately or can instruct the Window Manager to ignore palette activation and handle it themselves. For backward compatibility, Copland also supports applications that aren't Copland-savvy and patch `FrontWindow` to achieve appropriate window activation when tool palettes are present.

Moving Windows

Functions used for moving windows, such as `DragWindow` and `MoveWindow`, work much the same way in Copland as they do in System 7.

Resizing Windows

The Copland Window Manager supports window resizing in any direction. The window IDO specifies the grow direction, and you call the new

`ResizeWindow` function to track the mouse and resize the window as appropriate. The `ResizeWindow` function replaces the System 7 functions `GrowWindow` and `SizeWindow`, which are supported for backward compatibility only.

Zooming Windows

The Copland Window Manager extends the monitor-specific zooming behavior of Finder windows to all applications. Clicking the zoom box in a Copland window causes the window to expand so it zooms to its standard state (the maximum size of its content), if possible, without moving to a different monitor.

The new Window Manager function `ZoomWindowOut` zooms a window to its standard state. If the window can't fit at its ideal size, `ZoomWindowOut` zooms it to fit its current monitor.

The System 7 function `ZoomWindow` is still supported in Copland. In addition to zooming, `ZoomWindow` is useful for resizing a window programmatically in a direction other than down and to the right.

Collapsing Windows

System 7 supports window collapsing by means of the WindowShade extension, which collapses windows without applications' knowledge when the user double-clicks the window's title bar. This causes some compatibility problems for applications that cache certain information about a window's size and position.

Copland windows have a collapse box for collapsing windows. The version of the Copland Window Manager in this release provides a new `FindWindow` result code, `inCollapseBox`, that indicates a mouse-down event has occurred in the collapse box. You can respond to such an event by calling the new routine `CollapseWindow`. To determine whether a window is collapsed, you can call the `IsWindowCollapsed` function. As previously noted, the Copland event model and the Panels class library also make it possible to manipulate windows without calling the `FindWindow` function directly.

In applications that aren't Copland-savvy, windows that use standard system window definitions have collapse boxes and will be collapsed by the Window Manager without the application's knowledge.

Disposing of Windows

Copland-savvy applications should always use `DisposeWindow` to dispose of a window. The Copland Window Manager supports `CloseWindow` for backward compatibility only.

Maintaining the Update Region

The System 7 functions `InvalRgn`, `InvalRect`, `ValidRgn`, and `ValidRect` all assume that the current graphics port is actually a window pointer and adjust the update region of that window accordingly. Because the graphics port's pointer isn't the same as a Copland window reference, the Copland Window Manager replaces these functions with four new functions that take window references: `InvalWindowRgn`, `InvalWindowRect`, `ValidWindowRgn`, and `ValidWindowRect`.

The `BeginUpdate` and `EndUpdate` functions work much the same way in Copland as they do in System 7.

Setting and Retrieving Other Window Characteristics

The `SetWindowPic`, `SetWRefCon`, `GetWRefCon`, and `GetWVariant` functions work much the same way in Copland as they do in System 7.

Manipulating the Desktop

The `GetCWMgrPort` and `GetWMgrPort` functions are supported for backward compatibility only. They aren't recommended for Copland-savvy applications. The `SetDeskCPat` and `GetGrayRgn` functions aren't supported at all in Copland.

Manipulating Window Color Information

In Copland, window color information is handled entirely by the window IDO, which in turns relies on the current theme. The System 7 functions `SetWinColor` and `GetAuxWin` are supported only for backward compatibility, and (as in System 7) only with respect to background color.

Low-Level Routines

Low-level routines such as `ClipAbove`, `SaveOld`, and `DrawNew` are supported for backward compatibility only.

Window Manager Resource

The Copland Window Manager supports a new `'wind'` resource that Copland-savvy applications should use rather than the System 7 `'WIND'` resource, which is supported for backward compatibility only.

Dialog Manager, Control Manager, List Manager, Menu Manager

The interfaces for the Dialog Manager, Control Manager, List Manager, and Menu Manager in this release represent work in progress that will change significantly in later developer releases. The interfaces for the Panels class library in this release will be extended to provide equivalent services by means of panel attributes and methods.

The System 7 versions of the Dialog Manager, Control Manager, List Manager, and Menu Manager are supported for backward compatibility only. Copland-savvy applications must use the Panels class library to create dialog boxes, alert boxes, controls, lists, and menus.

For an introduction to panels, see Chapter 1, "Introduction to the Copland Toolbox."

Scrap Manager

The Scrap Manager used in System 7 has changed little since it was first created as part of the software for the original Macintosh computer. It was originally designed to handle a few lines of text or a 1-bit picture being copied and pasted between MacWrite and MacPaint, not the large pieces of data, such as QuickTime movies, sounds, and blocks of formatted text, commonly used today.

Copland replaces the original Scrap Manager with a new Clipboard Manager and introduces an entirely new Scrap Manager. The new Scrap Manager supplies the generic storage mechanism for copying and pasting Clipboard information and dragging and dropping data. Both the Clipboard Manager and the Drag Manager use the Scrap Manager to move data between clients (for

instance, between applications or within areas of a single application). The System 7 Drag Manager functions that prepared data for transport have been revised and incorporated into the Copland Scrap Manager so that they apply to Clipboard data as well as to drag information.

The Copland Scrap Manager allows you to create a scrap, add items to the scrap, and specify each scrap item with different scrap item types (that is, multiple representations). It also allows you to read and extract information from a scrap after it has been transported to its destination.

You can add collection items to a scrap at three levels: the scrap as a whole, individual items within the scrap, and scrap item types within each scrap item. For example, a collection item for the entire scrap might specify an identifier for the creator of the scrap; and a collection item at the scrap item level might indicate the order in which the user selected the items.

Many Scrap Manager, Clipboard Manager, and Drag Manger functions take a scrap reference as an input parameter. A scrap reference identifies a particular scrap, whether it is used by the Clipboard Manager, the Drag Manager, or the Scrap Manager.

Scrap Manager Functions

The Copland Scrap Manager provides functions you can use to create and delete scrap references, add items to the scrap, make and keep promises, obtain information about scrap items, and add collection items to a scrap.

Creating and Deleting Scrap References

You use the `NewScrapRef` function to create a new scrap and allocate a scrap reference for use with a Clipboard. The `DisposeScrapRef` function disposes of a scrap previously created by the `NewScrapRef` function.

Adding Scrap Items to the Scrap

The `AddScrapItemType` function, which replaces the System 7 Drag Manager function `AddDragItemFlavor`, lets you write data in a specific format to the scrap. You can use `AddScrapItemType` repeatedly to place data in more than one format in the scrap.

To add data to a specific item type, you can use the `SetScrapItemTypeData` function, which replaces the System 7 Drag Manager function `SetDragItemFlavorData`.

Making and Keeping Promises

The Copland Scrap Manager adds support for promises in Clipboard operations to the support provided by System 7 for promises in drag operations. You can use the `SetScrapSendProc` function to specify the scrap send function the Scrap Manager will use when a promise needs to be fulfilled. (This function replaces the System 7 Drag Manager `SetDragSendProc` function.) To specify the data fulfillment function for a scrap item type, you can use the Copland Scrap Manager function `SetScrapItemTypeDataFulfillmentProc`, which replaces the System 7 Drag Manager function `SetDragSendProc`.

Getting Scrap Item Information

The Copland Scrap Manager provides functions that obtain a range of data about scrap items, including the number of scrap items, the item reference number for a specified scrap item, the number of item types for a specified scrap item, the scrap item type associated with a particular location within a scrap item, the size of a specified scrap item type, and the data for a specified scrap item type. These scrap-item information functions replace the following System 7 Drag Manager functions: `CountDragItems`, `GetDragItemReferenceNumber`, `CountDragItemFlavors`, `GetFlavorType`, `GetFlavorDataSize`, and `GetFlavorData`.

Working With Collections

The Copland Scrap Manager includes a set of functions that let you retrieve add, or remove specific collection information about the entire scrap, a specific scrap item, and a scrap item type.

Clipboard Manager

The entirely new Copland Clipboard Manager supports cut, copy, and paste by accepting scraps created with the Scrap Manager and transferring them between clients via the familiar concept of the Clipboard.

Basically, the Copland Clipboard contains a single Scrap Manager scrap. Clients use the Scrap Manager to create the scrap, the Clipboard Manager to put it on or retrieve it from the Clipboard, and the Scrap Manager to read it. The Clipboard Manager accepts a Scrap Manager scrap, returns read access to a scrap on the Clipboard, and disposes of a scrap when a client is finished with it.

The Clipboard Manager manages the Clipboard. When you use the Clipboard Manager to put a scrap on the Clipboard, you can no longer write to that scrap (except for unfulfilled promises). The Clipboard Manager automatically disposes of the scrap when it has been replaced by a new scrap and other applications are finished retrieving it.

When using the Clipboard Manager to exchange data between applications or within your application, you follow these steps:

1. Create a scrap with the Scrap Manager's `NewScrapRef` function.
2. Add items to the scrap with the Scrap Manager's `AddScrapItemType` function.
3. Put the scrap on a Clipboard with the Clipboard Manager's `PutScrapOnClipboard` function.
4. When the user pastes, get the scrap from the Clipboard with the `GetClipboardScrapRef` function.
5. Obtain data from the scrap using the Scrap Manager functions.

The `GetClipboardScrapRef` function returns a Clipboard scrap reference that you can pass to Clipboard Manager functions and to Scrap Manager functions. The Copland Clipboard scrap reference is defined by the `ClipboardScrapRef` data type.

Clipboard Manager Functions

The Copland Clipboard Manager functions let you place a scrap on the Clipboard and retrieve a scrap from it.

Putting a Scrap on the Clipboard

The `PutScrapOnClipboard` function takes a scrap created using the Scrap Manager and puts it on the Clipboard. If this function succeeds, the scrap just written becomes the active Clipboard scrap and the Clipboard Manager disposes of the scrap when it's finished with it. If the function fails, the client must dispose of the scrap.

Retrieving and Releasing a Scrap From the Clipboard

The `GetClipboardScrap` function retrieves a read-only copy of the scrap from the Clipboard. After retrieving a scrap from the Clipboard, you use Scrap Manager functions to extract its data. When you're finished with a scrap retrieved from the Clipboard, you use the `ReleaseClipboardScrap` function to release the scrap reference.

Drag Manager

The System 7 Drag Manager supports all aspects of drag and drop behavior, both in the Finder and within applications. The Copland Drag Manager supports the user experience of dragging, but it no longer packs and unpacks the data that's transported in a drag. Instead, you use the Drag Manager to create the scrap, the Scrap Manager to add items to it, the Drag Manager to support the user experience during dragging, and the Scrap Manager to read the scrap after the drag operation is complete.

As mentioned in "Scrap Manager," beginning on page 2-22, the System 7 Drag Manager functions that prepare data for transport have been revised and incorporated into the Copland Scrap Manager so that they apply to Clipboard data as well as to drag information. The Copland Drag Manager differs from the System 7 Drag Manager in several other important respects:

- As the user drags an image around the screen, the Copland Drag Manager displays either a transparent version of the original image or just its outline.

- The Appearance Manager chooses the highlighting colors (for example, the target frame color) for drag operations.
- To add or obtain data from a drag, you use the Scrap Manager and pass the `DragScrapRef` data type directly.
- You no longer use the Drag Manager data type `FlavorFlags`. Instead, the Scrap Manager provides the equivalent type `ScrapItemTypeFlags`, used in collection items that you attach to a drag scrap using the Scrap Manager. Similarly, the System 7 data types `HFSFlavor` and `PromiseHFSFlavor` have been transferred to the Scrap Manager under new names.
- You use the Scrap Manager data type `ScrapItemType` instead of the System 7 data type `DragItemFlavor`.
- The System 7 Drag Manager `ItemReference` data type has been renamed `ScrapItemRef`, and the `FlavorType` data type has been renamed `ScrapItemType`. Their uses remain exactly the same.
- The following System 7 Drag Manager functions have been replaced by parallel functions in the Copland Scrap Manager: `AddDragItemFlavor`, `SetDragItemFlavorData`, `SetDragSendProc`, `CountDragItems`, `GetDragItemReferenceNumber`, `CountDragItemFlavors`, `GetFlavorType`, `GetFlavorFlags`, `GetFlavorDataSize`, and `GetFlavorData`. Copland-savvy applications should not use these functions.

When you use the Drag Manager to perform a drag operation, you follow these steps:

1. Create a drag scrap with the `NewDrag` function.
2. Add items to the scrap with the Scrap Manager's `AddScrapItemType` function.
3. Perform a single drag operation using the `TrackDrag` function. (This operation can be canceled.)
4. When the user drops the dragged item, the Drag Manager uses your receive drag handler to receive the drag.
5. Use Scrap Manager functions to retrieve data from the scrap.

Most Drag Manager functions take a drag scrap reference as an input parameter. A drag scrap reference can also be passed to Scrap Manager functions. A drag scrap reference must be allocated by the Drag Manager function `NewDrag` and is defined by the `DragScrapRef` data type.

Drag Manager Functions

The Copland Drag Manager functions let you install and remove drag handler functions, create and dispose of drag scrap references, override standard input and drawing behavior, perform a drag, set the transparent drag image, set and get status information about a drag, and support drag and drop behavior.

Installing and Removing Drag Handler Functions

In Copland, you still use the Drag Manager to install or remove drag handler functions for your entire application or for one of your application's windows. The Drag Manager provides a pair of install and remove functions for the drag tracking handler and the drag receive handler, which are defined by the `DragTrackingHandler` and `DragReceiveHandler` data types.

Creating and Disposing of Drag Objects and References

The Copland version of the `NewDrag` function creates a drag scrap reference to identify the drag in subsequent calls to the Drag Manager. This drag scrap reference is required when you add scrap item types via the Scrap Manager and when you call the `TrackDrag` function. Your installed drag handlers receive this drag scrap reference so that you can call other Drag Manager functions within your drag handlers.

The Copland version of the `DisposeDrag` function disposes of the drag scrap identified by a specified drag scrap reference. If the drag scrap contains any scrap item types, the memory associated with the scrap item types is disposed of as well. You should call `DisposeDrag` after a drag has been performed using `TrackDrag` or to dispose of a drag scrap reference that's no longer needed.

Overriding Standard Input and Drawing Behavior

If you want to override the Copland Drag Manager's default behavior, you can provide drag callback functions for drag input and drag drawing with the `SetDragInputProc` and `SetDragDrawingProc` functions. The System 7 `SetDragSendProc` callback function has been replaced by the Copland Scrap Manager function `SetScrapItemTypeDataFulfillmentProc`.

Performing a Drag

Once the drag image for a drag has been set up, you use the Copland version of the `TrackDrag` function to perform the drag operation with a particular drag

scrap reference given a mouse-down event and drag region. The Drag Manager follows the cursor on the screen with the specified drag image feedback and sends tracking messages to applications with registered drag tracking handlers. When the user releases the mouse button, the Drag Manager calls any receive drop handlers registered for the destination window. An application's receive drop handler accepts the drag and transfers the dragged data into the application.

Setting the Transparency of the Drag Image

You can use the new `SetDragImage` function to set the degree of transparency of a given drag image.

Supporting Drag and Drop Behavior

Like the System 7 Drag Manager, the Copland Drag Manager supports drag and drop behavior with functions that retrieve and set an Apple event descriptor for a specified drop location, perform standard drag and drop highlighting (including scrolling preparation), and let you draw zooming animation like the Finder's.

Getting and Setting Status Information About a Drag

The Copland Drag Manager includes functions that obtain status information about the drag attribute flags, get and set the mouse location, retrieve the origin of a specified drag, obtain key modifiers associated with a specified drag scrap reference, and set and retrieve the bounding rectangle of specified drag items.

Resource Manager

Most of the System 7 Resource Manager functions are supported for backward compatibility. The exceptions are the `InitResources`, `RsrcZoneInit`, and `RsrcMapEntry` functions, which even System 7 applications don't need to call. Also, the undocumented resource chain override mechanism used by some System 7 applications is not supported in Copland.

Most of the System 7 functions also have an equivalent function, whose name begins with the prefix `RM`, for use by Copland-savvy applications. Major

differences between the new functions and System 7 functions include the following:

- You can't access the resource map in Copland. Copland supports an opaque resource file abstraction, and you access its resources through this abstraction.
- The error mechanism based on `ResError` is no longer necessary. Instead, Copland Resource Manager functions simply return `OSStatus` errors.
- The `FSpCreateResFile`, `HCreateResFile`, and `CreateResFile` functions have been replaced by the single function `RMCreateResFile`, which takes a Copland file system object.
- The `FSpOpenResFile`, `HOpenResFile`, `OpenRFPerm`, and `OpenResFile` functions have been replaced by the single function `RMOpenResFile`, which takes a Copland file system object.
- The new functions `RMAddResFileToSearchPath` and `RMRemoveResFileFromSearchPath` allow you to add or remove resource files to or from the beginning of the resource search path for your application.
- Functions such as `Get1Resource` and `Get1NamedResource` are no longer needed. Instead, you specify parameters for `RMGetResource`, `RMGetNamedResource`, and so on that indicate whether or not the function should search just the current resource file or the entire resource search path.
- Copland doesn't support ROM-based resources, so there is no Copland equivalent to the `RGetResource` function.
- The new Resource Manager functions include a parameter that allows you to enable or disable automatic loading of resource data into memory. As a result, there is no need for a Copland equivalent of the `SetResLoad` function.
- The System 7 function `GetResourceSizeOnDisk` has been renamed `RMGetResourceSize`.
- There is no Copland equivalent to the `GetMaxResourceSize` function, because using them depends on specific implementation details of the System 7 Resource Manager.
- The `GetResFileAttrs` and `SetResFileAttrs` functions have been replaced by the `GetResFileReadOnlyState` and `SetResFileReadOnlyState` functions, which get and set the resource file's read-only state both in memory and on disk.